

Java

РЕШЕНИЕ ПРАКТИЧЕСКИХ ЗАДАЧ

СОВЕРШЕНСТВУЙТЕ СВОИ НАВЫКИ, РЕШАЯ ПОВСЕДНЕВНЫЕ ТИПОВЫЕ ЗАДАЧИ



Анджел Леонард

Packt>

www.packt.com



Java Coding Problems

Improve your Java Programming skills by solving
real-world coding challenges

Anghel Leonard

Packt>

BIRMINGHAM - MUMBAI

Анджел Леонард

Java

РЕШЕНИЕ ПРАКТИЧЕСКИХ ЗАДАЧ

Санкт-Петербург

«БХВ-Петербург»

2021

УДК 004.438
ББК 32.973-018.1
Л47

Леонард А.

Л47 Java. Решение практических задач : пер. с англ. — СПб.: БХВ-Петербург, 2021. — 720 с.: ил.

ISBN 978-5-9775-6719-0

Рассмотрены задачи, встречающиеся в повседневной работе любого разработчика в среде Java. Приведено более 300 приложений, содержащих свыше 1000 примеров. Продемонстрированы эффективные практические приемы и технические решения с учетом сложности кода, производительности, удобочитаемости и многого другого.

Рассмотрены строки, числа, объекты, массивы, коллекции и структуры данных, работа с датой и временем. Приведены задачи на логический вывод типов, а также файловый ввод/вывод. Представлены задачи, связанные с API рефлексии Java. Особое внимание уделено программированию в функциональном стиле, задачам с привлечением конкурентности, правилам работы с классом Optional, а также API HTTP-клиента и API протокола WebSocket.

Для Java-программистов

УДК 004.438
ББК 32.973-018.1

Группа подготовки издания:

Руководитель проекта	<i>Евгений Рыбаков</i>
Зав. редакцией	<i>Людмила Гауль</i>
Перевод с английского	<i>Андрея Логунова</i>
Компьютерная верстка	<i>Натали Смирновой</i>
Оформление обложки	<i>Карины Соловьевой</i>

© Packt Publishing 2019. First published in the English language under the title 'Java Coding Problems – (9781789801415)'

Впервые опубликовано на английском языке под названием 'Java Coding Problems – (9781789801415)'

Подписано в печать 30 04 21
Формат 70×100^{1/16}. Печать офсетная. Усл печ л 58,05
Тираж 1200 экз. Заказ № 8321
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20



Отпечатано в ПАО «Можайский полиграфический комбинат»
143200, Россия, г. Можайск, ул. Мира, 93
www.oao.mpk.ru, тел. (495) 745-84-28, (49638) 20-685

ISBN 978-1-78847-141-5 (англ.)
ISBN 978-5-9775-6719-0 (рус.)

© Packt Publishing 2019
© Перевод на русский язык, оформление
ООО "БХВ-Петербург", ООО "БХВ", 2021

Оглавление

Составители	19
Об авторе	19
О рецензентах	19
Предисловие	21
Для кого эта книга предназначена	21
Что эта книга охватывает	22
Как получить максимальную отдачу от этой книги	25
Файлы с примерами исходного кода	25
Цветные изображения	26
Код в действии	26
Условные обозначения	26
Комментарии переводчика	27
Глава 1. Строки, числа и математика	29
Задачи	29
Решения	32
1. Подсчет повторяющихся символов	32
О символах Юникода	33
2. Отыскание первого неповторяющегося символа	36
3. Инвертирование букв и слов	38
4. Проверка, содержит ли строковое значение только цифры	39
5. Подсчет гласных и согласных	40
6. Подсчет появлений некоторого символа	42
7. Конвертирование строки в значение типа <i>int</i> , <i>long</i> , <i>float</i> или <i>double</i>	43
8. Удаление пробелов из строки	44
9. Соединение нескольких строк с помощью разделителя	44
10. Генерирование всех перестановок	46
11. Проверка, что строка является палиндромом	48
12. Удаление повторяющихся символов	49
13. Удаление заданного символа	51
14. Отыскание символа с наибольшим числом появлений	52
15. Сортировка массива строк по длине	54
16. Проверка наличия подстроки в строке	56
17. Подсчет числа появлений подстроки в строке	56

18. Проверка, являются ли две строки анаграммами	58
19. Объявление многострочных строковых литералов (или текстовых блоков).....	59
20. Конкатенирование одной и той же строки <i>n</i> раз.....	60
21. Удаление начальных и замыкающих пробелов	62
22. Поиск наибольшего общего префикса.....	62
23. Применение отступа.....	63
24. Трансформирование строк.....	65
25. Вычисление минимума и максимума двух чисел.....	66
26. Сложение двух крупных чисел типа <i>int/long</i> и переполнение операции.....	67
27. Строка как беззнаковое число с основанием системы счисления	68
28. Конвертирование в число посредством беззнаковой конверсии	69
29. Сравнение двух беззнаковых чисел	70
30. Вычисление частного и остатка от деления беззнаковых значений.....	70
31. Значение типа <i>double/float</i> является конечным значением с плавающей точкой	71
32. Применение логического И/ИЛИ/исключающего ИЛИ к двум булевым выражениям.....	72
33. Конвертирование значения типа <i>BigInteger</i> в примитивный тип.....	73
34. Конвертирование значение типа <i>long</i> в значение типа <i>int</i>	74
35. Вычисление целой части деления и целой части остатка от деления	75
36. Следующее значение с плавающей точкой.....	76
37. Умножение двух крупных значений типа <i>int/long</i> и переполнение операции.....	77
38. Совмещенное умножение-сложение (Fused Multiply Add).....	79
39. Компактное форматирование чисел	79
Форматирование	80
Разбор числа	82
Резюме.....	83
Глава 2. Объекты, немутуируемость и выражения <i>switch</i>	84
Задачи.....	84
Решения.....	86
40. Проверка ссылок на <i>null</i> в функциональном и императивном стилях программирования.....	86
41. Проверка ссылок на <i>null</i> и выбрасывание собственного исключения <i>NullPointerException</i>	89
42. Проверка ссылок на <i>null</i> и выбрасывание заданного исключения	91
43. Проверка ссылок на <i>null</i> и возврат непустых ссылок, заданных по умолчанию.....	93
44. Проверка индекса в интервале от 0 до длины.....	94
45. Проверка подынтервала в интервале от 0 до длины	96
46. Методы <i>equals()</i> и <i>hashCode()</i>	97
47. Немутуируемые объекты в двух словах	103
48. Немутуируемая строка.....	104
Достоинства немутуируемости строк	104
Недостатки немутуируемости строк	106
Является ли класс <i>String</i> полностью немутуируемым?	107

49. Немутулируемый класс.....	108
50. Передача мутулируемых объектов в немутулируемый класс и возврат мутулируемых объектов из него	109
51. Написание немутулируемого класса с помощью шаблона строителя.....	112
52. Предотвращение плохих данных в немутулируемых объектах	114
53. Клонирование объектов.....	116
Клонирование ручное	117
Клонирование методом <i>clone()</i>	117
Клонирование посредством конструктора	118
Клонирование посредством библиотеки <i>Cloning</i>	119
Клонирование посредством сериализации	120
Клонирование посредством JSON.....	121
54. Переопределение метода <i>toString()</i>	121
55. Выражения <i>switch</i>	123
56. Многочисленные метки вариантов	126
57. Блоки инструкций.....	126
Резюме.....	127
Глава 3. Работа с датой и временем.....	128
Задачи.....	128
Решения.....	130
58. Конвертирование строки в дату и время	130
До JDK 8	130
Начиная с JDK 8.....	130
59. Форматирование даты и времени.....	134
60. Получение текущих даты и времени без времени или даты	137
61. Класс <i>LocalDateTime</i> из <i>LocalDate</i> и <i>LocalTime</i>	137
62. Машинное время посредством класса <i>Instant</i>	137
Конвертирование значения <i>String</i> в объект <i>Instant</i>	138
Прибавление времени к объекту <i>Instant</i> или вычитание времени из объекта <i>Instant</i>	138
Сравнение объектов <i>Instant</i>	139
Конвертирование из объекта <i>Instant</i> в <i>LocalDateTime</i> , <i>ZonedDateTime</i> и <i>OffsetDateTime</i> и обратно	139
63. Определение временного периода с использованием значений на основе даты и значений на основе времени	140
Период времени с использованием значений на основе даты.....	140
Продолжительность времени с использованием значений на основе времени	142
64. Получение единиц даты и времени.....	144
65. Прибавление к дате и времени и вычитание из них.....	145
Работа с датой	145
Работа с <i>LocalDateTime</i>	145
66. Получение всех часовых поясов в UTC и GMT.....	146
До JDK 8	147
Начиная с JDK 8.....	147

67. Получение локальных даты и времени во всех имеющихся часовых поясах...	148
До JDK 8	149
Начиная с JDK 8.....	149
68. Вывод на экран информации о дате и времени полета	150
69. Конвертирование временной метки UNIX в дату и время	151
70. Отыскание первого/последнего дня месяца.....	152
71. Определение/извлечение поясных смещений.....	154
До JDK 8	155
Начиная с JDK 8.....	155
72. Конвертирование из даты во время и наоборот.....	156
<i>Date</i> — <i>Instant</i>	156
<i>Date</i> — <i>LocalDate</i>	157
<i>Date</i> — <i>DateLocalTime</i>	158
<i>Date</i> — <i>ZonedDateTime</i>	158
<i>Date</i> — <i>OffsetDateTime</i>	159
<i>Date</i> — <i>LocalTime</i>	159
<i>Date</i> — <i>OffsetTime</i>	159
73. Обход интервала дат в цикле.....	160
До JDK 8	160
Начиная с JDK 8.....	160
Начиная с JDK 9.....	161
74. Вычисление возраста.....	161
До JDK 8	161
Начиная с JDK 8.....	161
75. Начало и конец дня.....	162
76. Разница между двумя датами	164
До JDK 8	164
Начиная с JDK 8.....	165
77. Имплементация шахматных часов.....	166
Резюме.....	171
Глава 4. Логический вывод типов	172
Задачи.....	172
Решения.....	174
78. Простой пример с использованием переменной типа <i>var</i>	174
79. Использование типа <i>var</i> с примитивными типами.....	177
80. Использование типа <i>var</i> и неявного приведения типов для поддержания технической сопровождаемости исходного кода	178
81. Явное понижающее приведение типов или как избежать типа <i>var</i>	180
82. Как отказаться от типа <i>var</i> , если вызываемые имена не содержат для людей достаточной информации о типе.....	180
83. Сочетание LVTI и программирования согласно интерфейсу.....	182
84. Сочетание LVTI и ромбовидного оператора.....	183
85. Присвоение массива переменной типа <i>var</i>	184
86. Использование LVTI в составных объявлениях	184
87. LVTI и область видимости переменной	185

88. LVTI и тернарный оператор	186
89. LVTI и циклы <i>for</i>	187
90. LVTI и потоки.....	188
91. Использование LVTI для разбиения вложенных/крупных цепочек выражений.....	189
92. LVTI и возвращаемый и аргументный типы метода.....	190
93. LVTI и анонимные классы.....	191
94. LVTI может быть финальным и практически финальным	192
95. LVTI и лямбда-выражения	193
96. LVTI и инициализаторы <i>null</i> , экземплярные переменные и переменные блоков <i>catch</i>	194
Инструкция <i>try</i> с объявлением ресурса.....	194
97. LVTI и обобщенные типы <i>T</i>	195
98. LVTI, подстановочные знаки, коварианты и контраварианты.....	196
LVTI и подстановочные знаки.....	196
LVTI и коварианты/контраварианты	197
Резюме.....	198
Глава 5. Массивы, коллекции и структуры данных.....	199
Задачи.....	199
Решения.....	201
99. Сортировка массива	201
Решения, встроенные в JDK	202
Другие алгоритмы сортировки	204
100. Поиск элемента в массиве.....	213
Проверка только присутствия элемента	214
Проверка только первого индекса.....	217
101. Проверка эквивалентности или несовпадения двух массивов	218
Проверка эквивалентности двух массивов.....	218
Проверка несовпадения в двух массивах.....	220
102. Сравнение двух массивов лексикографически	221
103. Создание потока из массива	223
104. Минимальное, максимальное и среднее значения массива.....	224
Вычисление максимума и минимума.....	225
Вычисление среднего значения.....	227
105. Инвертирование массива	227
106. Заполнение и настройка массива	229
107. Следующий больший элемент.....	231
108. Изменение размера массива	232
109. Создание немодифицируемых/немутируемых коллекций	233
Задача 1 (<i>Collections.unmodifiableList()</i>).....	233
Задача 2 (<i>Arrays.asList()</i>)	234
Задача 3 (<i>Collections.unmodifiableList()</i> и статический блок).....	234
Задача 4 (<i>List.of()</i>).....	235
Задача 5 (немутируемое)	236
110. Возврат значения по умолчанию из коллекции <i>Map</i>	238

111. Вычисление отсутствия/присутствия элемента в отображении <i>Map</i>	239
Пример 1 (<i>computeIfPresent()</i>)	239
Пример 2 (<i>computeIfAbsent()</i>)	240
Пример 3 (<i>compute()</i>)	241
Пример 4 (<i>merge()</i>)	243
Пример 5 (<i>putIfAbsent()</i>)	244
112. Удаление элемента из отображения <i>Map</i>	244
113. Замена элементов в отображении <i>Map</i>	245
114. Сравнение двух отображений <i>Map</i>	246
115. Сортировка отображения <i>Map</i>	248
Сортировка по ключу посредством <i>TreeMap</i> и в естественном порядке	249
Сортировка по ключу и значению посредством <i>Stream</i> и <i>Comparator</i>	249
Сортировка по ключу и значению посредством <i>List</i>	250
116. Копирование отображения <i>HashMap</i>	251
117. Слияние двух отображений <i>Map</i>	252
118. Удаление всех элементов коллекции, которые совпадают с предикатом	253
Удаление посредством итератора	254
Удаление посредством <i>Collection.removeIf()</i>	254
Удаление посредством <i>Stream</i>	254
Разбиение элементов посредством коллектора <i>Collectors.partitioningBy()</i>	255
119. Конвертирование коллекции в массив	255
120. Фильтрация коллекции по списку	256
121. Замена элементов в списке	258
122. Нитебезопасные коллекции, стеки и очереди	259
Конкурентные коллекции	259
Конкурентные коллекции против синхронизированных	264
123. Поиск сперва в ширину	264
124. Префиксное дерево	266
Вставка слова в префиксное дерево	268
Отыскание слова в префиксном дереве	269
Удаление слова из префиксного дерева	269
125. Кorteж	271
126. Структура данных Union Find	272
Имплементирование операции поиска	275
Имплементирование операции объединения	276
127. Дерево Фенвика, или двоичное индексированное дерево	277
128. Фильтр Блума	281
Резюме	284
Глава 6. Пути, файлы, буферы, сканирование и форматирование ввода/вывода в среде Java	285
Задачи	285
Решения	287
129. Создание путей к файлам	287
Создание пути относительно корня хранилища файлов	288
Создание пути относительно текущей папки	288

Создание абсолютного пути	289
Создание пути с помощью сокращений	289
130. Конвертирование путей к файлам	291
131. Присоединение путей к именам файлов	292
132. Конструирование пути между двумя местоположениями	293
133. Сравнение путей к файлам	294
Метод <i>Path.equals()</i>	294
Пути, представляющие одинаковый файл/папку	295
Лексикографическое сравнение	295
Частичное сравнение	296
134. Прохождение путей	296
Тривиальный обход папки	297
Поиск файла по имени	297
Удаление папки	299
Копирование папки	301
Метод <i>Files.walk()</i> JDK 8	303
135. Наблюдение за путями	304
Отслеживание изменений папки	305
136. Поточковая передача содержимого файла	307
137. Поиск файлов/папок в дереве файлов	308
138. Эффективное чтение/запись текстовых файлов	310
Чтение текстовых файлов в память	314
Запись текстовых файлов	316
139. Эффективное чтение/запись двоичных файлов	317
Чтение двоичных файлов в память	319
Запись двоичных файлов	320
140. Поиск в больших файлах	321
Решение на основе класса <i>BufferedReader</i>	322
Решение на основе метода <i>Files.readAllLines()</i>	322
Решение на основе метода <i>Files.lines()</i>	323
Решение на основе класса <i>Scanner</i>	323
Решение на основе класса <i>MappedByteBuffer</i>	324
141. Чтение файла JSON/CSV как объекта	325
Чтение/запись файла JSON как объекта	325
Чтение CSV-файла как объекта	328
142. Работа с временными файлами/папками	330
Создание временных папки или файла	330
Удаление временной папки или временного файла посредством перехватчика отключения	331
Удаление временной папки или временного файла посредством метода <i>deleteOnExit()</i>	332
Удаление временного файла посредством аргумента <i>DELETE_ON_CLOSE</i>	333
143. Фильтрация файлов	334
Фильтрация посредством метода <i>Files.newDirectoryStream()</i>	334
Фильтрация посредством интерфейса <i>FilenameFilter</i>	336
Фильтрация посредством интерфейса <i>FileFilter</i>	337

144. Обнаружение несовпадений между двумя файлами	337
145. Кольцевой байтовый буфер	339
146. Лексемизация файлов	347
147. Запись форматированного вывода непосредственно в файл	351
148. Работа с классом <i>Scanner</i>	354
<i>Scanner</i> против <i>BufferedReader</i>	357
Резюме	358
Глава 7. Классы, интерфейсы, конструкторы, методы и поля в API рефлексии Java	359
Задачи	359
Решения	361
149. Инспектирование пакетов	361
Получение классов пакета	362
Инспектирование пакетов внутри модулей	365
150. Инспектирование классов	365
Получение имени класса <i>Pair</i> посредством экземпляра	366
Получение модификаторов класса <i>Pair</i>	366
Получение имплементированных интерфейсов класса <i>Pair</i>	366
Получение конструкторов класса <i>Pair</i>	367
Получение полей класса <i>Pair</i>	367
Получение методов класса <i>Pair</i>	368
Получение модуля класса <i>Pair</i>	369
Получение надкласса класса <i>Pair</i>	369
Получение имени некоторого типа	369
Получение строк, описывающих класс	370
Получение строки с дескриптором типа для класса	370
Получение типа компонента для массива	370
Получение класса для индексируемого типа, тип компонента которого описывается классом <i>Pair</i>	371
151. Создание экземпляра класса посредством отрефлексированного конструктора	371
Создание экземпляра класса посредством приватного конструктора	372
Создание экземпляра класса из файла JAR	373
Полезные фрагменты кода	374
152. Извлечение аннотации типа получателя	374
153. Получение синтетических и мостовых конструкций	376
154. Проверка переменного числа аргументов	377
155. Проверка методов по умолчанию	378
156. Управление гнездовым доступом посредством рефлексии	378
Доступ посредством API рефлексии	380
157. Рефлексия для геттеров и сеттеров	382
Получение геттеров и сеттеров	382
Генерирование геттеров и сеттеров	385
158. Рефлексиование аннотаций	389
Инспектирование аннотаций пакетов	389
Инспектирование аннотаций классов	390

Инспектирование аннотаций методов	390
Инспектирование аннотаций выбрасываемых исключений	391
Инспектирование аннотаций типов значений, возвращаемых из методов	392
Инспектирование аннотаций параметров методов	392
Инспектирование аннотаций полей	393
Инспектирование аннотаций надклассов	393
Инспектирование аннотаций интерфейсов.....	394
Получение аннотаций по типу.....	394
Получение объявленной аннотации	395
159. Инициирование экземплярного метода.....	395
160. Получение статических методов.....	396
161. Получение обобщенных методов, полей и исключений.....	397
Обобщения методов.....	398
Обобщения полей.....	399
Обобщения надклассов.....	399
Обобщения интерфейсов.....	399
Обобщения исключений.....	399
162. Получение публичных и приватных полей.....	400
163. Работа с массивами.....	401
164. Инспектирование модулей.....	402
165. Динамические посредники	404
Имплементирование динамического посредника.....	405
Резюме.....	407
Глава 8. Программирование в функциональном стиле —	
основы и шаблоны архитектурного дизайна.....	408
Задачи.....	408
Решения.....	409
166. Написание функциональных интерфейсов	409
День 1 (фильтрация дынь по их сорту)	410
День 2 (фильтрация дынь по их весу)	411
День 3 (фильтрация дынь по их сорту и весу)	411
День 4 (передача поведения в качестве параметра).....	412
День 5 (имплементирование еще 100 фильтров)	414
День 6 (анонимные классы могут быть написаны как лямбда-выражения)...	415
День 7 (абстрагирование от типа <i>List</i>)	415
167. Лямбда-выражение в двух словах	417
168. Имплементирование шаблона исполнения вокруг опорной операции	418
169. Имплементирование шаблона фабрики.....	420
170. Имплементирование шаблона стратегий	422
171. Имплементирование шаблона трафаретного метода	423
172. Имплементирование шаблона наблюдателя	425
173. Имплементирование шаблона одалживания.....	428
174. Имплементирование шаблона декоратора	430
175. Имплементирование шаблона каскадного строителя	434
176. Имплементирование шаблона команд.....	435
Резюме.....	438

Глава 9. Программирование в функциональном стиле — глубокое погружение	439
Задачи	439
Решения	441
177. Тестирование функций высокого порядка	441
Тестирование метода, принимающего лямбда-выражение в качестве параметра	441
Тестирование метода, возвращающего функциональный интерфейс	442
178. Тестирование методов, использующих лямбда-выражения	443
179. Отладка лямбда-выражений	445
180. Фильтрация непустых элементов потока	448
181. Бесконечные потоки, <i>takeWhile()</i> и <i>dropWhile()</i>	450
Бесконечный последовательный упорядоченный поток	451
Нелимитированный поток псевдослучайных значений	453
Бесконечный последовательный неупорядоченный поток	454
Брать до тех пор, пока предикат возвращает <i>true</i>	454
Отбрасывать до тех пор, пока предикат возвращает <i>true</i>	456
182. Отображение элементов потока в новый поток	458
Использование метода <i>Stream.map()</i>	458
Использование метода <i>Stream.flatMap()</i>	460
183. Отыскание элементов в потоке	463
Метод <i>findAny</i>	464
Метод <i>findFirst</i>	464
184. Сопоставление элементов в потоке	465
185. Сумма, максимум и минимум в потоке	466
Терминальные операции <i>sum()</i> , <i>min()</i> и <i>max()</i>	467
Редукция	468
186. Сбор результатов потока	470
187. Соединение результатов потока	473
188. Подытоживающие коллекторы	474
Суммирование	474
Усреднение	476
Подсчет	476
Максимум и минимум	477
Все вместе	477
189. Группирование	478
Одноуровневое группирование	478
Многоуровневое группирование	484
190. Разбиение	486
191. Фильтрующий, сглаживающий и отображающий коллекторы	488
Коллектор <i>filtering()</i>	489
Коллектор <i>mapping()</i>	490
Коллектор <i>flatMapMapping()</i>	490
192. Сочленение	492
193. Написание собственного коллектора	494
Поставщик <i>Supplier<A> supplier()</i> ;	496

Накопление элементов — <i>BiConsumer<A, T> accumulator()</i> ;	497
Применение финального преобразования — <i>Function<A, R> finisher()</i> ;	497
Параллелизация коллектора — <i>BinaryOperator<A> combiner()</i> ;	497
Возврат финального результата — <i>Function<A, R> finisher()</i> ;	498
Характеристики — <i>Set<Characteristics> characteristics()</i> ;	498
Проверка времени	498
Собственный коллектор посредством метода <i>collect()</i>	499
194. Ссылка на метод	499
Ссылка на статический метод	500
Ссылка на экземплярный метод	500
Ссылка на конструктор	501
195. Параллельная обработка потоков	501
Сплитераторы	504
Написание собственного сплитератора	506
196. <i>Null</i> -безопасные потоки	506
197. Композиция функций, предикатов и компараторов	508
Композиция предикатов	509
Композиция компараторов	510
Композиция функций	512
198. Методы по умолчанию	513
Резюме	515

Глава 10. Конкурентность — пулы нитей исполнения, объекты <i>Callable</i> и синхронизаторы	516
Задачи	516
Решения	518
199. Состояния жизненного цикла нити исполнения	518
Состояние <i>NEW</i>	519
Состояние <i>RUNNABLE</i>	519
Состояние <i>BLOCKED</i>	520
Состояние <i>WAITING</i>	521
Состояние <i>TIMED_WAITING</i>	522
Состояние <i>TERMINATED</i>	523
200. Использование замка на уровне объекта против использования на уровне класса	524
Замок на уровне объекта	524
Замок на уровне класса	525
Важно знать	526
201. Пулы нитей исполнения в среде Java	527
Интерфейс <i>Executor</i>	527
Интерфейс <i>ExecutorService</i>	527
Интерфейс <i>ScheduledExecutorService</i>	530
Пулы нитей исполнения посредством класса <i>Executors</i>	530
202. Пул нитей исполнения с одной нитью	531
Производитель ждет, когда потребитель освободится	532
Производитель не ждет, когда потребитель освободится	537

203. Пул нитей исполнения с фиксированным числом нитей.....	539
204. Пулы кэшированных и запланированных нитей исполнения	540
205. Пул обкрадывающих нитей исполнения	547
Большое число мелких операций	549
Малое число времязатратных операций	551
206. Операции <i>Callable</i> и <i>Future</i>	553
Отмена операции <i>Future</i>	557
207. Инициирование нескольких операций <i>Callable</i>	558
208. Стопоры	560
209. Барьер	564
210. Обменник	568
211. Семафоры	571
212. Фазировщики	574
Резюме	579
Глава 11. Конкурентность — глубокое погружение.....	580
Задачи	580
Решения	581
213. Прерываемые методы	581
214. Каркас разветвления/соединения	585
Вычисление суммы посредством класса <i>RecursiveTask</i>	587
Вычисление чисел Фибоначчи посредством класса <i>RecursiveAction</i>	588
Использование класса <i>CountedCompleter</i>	590
215. Каркас разветвления/соединения метод <i>compareAndSetForkJoinTaskTag()</i>	593
216. Класс <i>CompletableFuture</i>	596
Выполнение асинхронных операций и возврат <i>void</i>	596
Выполнение асинхронной операции и возврат ее результата	597
Выполнение асинхронной операции и возврат результата посредством явного пула нитей исполнения	598
Прикрепление функции обратного вызова, которая обрабатывает результат асинхронной операции и возвращает результат	598
Прикрепление функции обратного вызова, которая обрабатывает результат асинхронной операции и возвращает <i>void</i>	600
Прикрепление функции обратного вызова, которая выполняется после асинхронной операции и возвращает <i>void</i>	601
Обработка исключений асинхронной операции посредством <i>exceptionally()</i>	602
Метод <i>exceptionallyCompose()</i> JDK 12	607
Обработка исключений асинхронной операции посредством метода <i>handle()</i>	608
Явное завершение экземпляра класса <i>CompletableFuture</i>	610
217. Сочетание нескольких экземпляров класса <i>CompletableFuture</i>	611
Сочетание посредством метода <i>thenCompose()</i>	611
Сочетание посредством метода <i>thenCombine()</i>	612
Сочетание посредством метода <i>allOf()</i>	613
Сочетание посредством метода <i>anyOf()</i>	615
218. Оптимизирование занятого ожидания	616

218. Оптимизирование занятого ожидания.....	616
219. Отмена операции.....	617
220. Переменные типа <i>ThreadLocal</i>	618
Экземпляры в расчете на нить исполнения.....	619
Контекст в расчете на нить исполнения.....	621
221. Атомарные переменные.....	623
Сумматоры и накопители.....	626
222. Класс <i>ReentrantLock</i>	627
223. Класс <i>ReentrantReadWriteLock</i>	630
224. Класс <i>StampedLock</i>	633
225. Тупик (обед философов).....	637
Резюме.....	640
Глава 12. Класс <i>Optional</i>.....	641
Задачи.....	641
Решения.....	643
226. Инициализация объекта класса <i>Optional</i>	644
227. Метод <i>Optional.get()</i> и недостающее значение.....	644
228. Возврат уже сконструированного значения по умолчанию.....	645
229. Возврат несуществующего значения по умолчанию.....	646
230. Выбрасывание исключения <i>NoSuchElementException</i>	647
231. Объект класса <i>Optional</i> и ссылки <i>null</i>	648
232. Потребление присутствующего объекта класса <i>Optional</i>	649
233. Возврат присутствующего объекта класса <i>Optional</i> или еще одного.....	650
234. Выстраивание лямбда-выражений в цепочку посредством метода <i>orElseFoo()</i>	651
235. Нежелательное использование типа <i>Optional</i> только для получения значения.....	652
236. Нежелательное использование типа <i>Optional</i> для полей.....	653
237. Нежелательное использование типа <i>Optional</i> в аргументах конструктора.....	654
238. Нежелательное использование типа <i>Optional</i> в аргументах сеттера.....	655
239. Нежелательное использование типа <i>Optional</i> в аргументах метода.....	656
240. Нежелательное использование типа <i>Optional</i> для возврата пустых либо <i>null</i> -коллекций или массивов.....	658
241. Как избежать использования типа <i>Optional</i> в коллекциях.....	659
242. Путаница метода <i>of()</i> с методом <i>ofNullable()</i>	660
243. <i>Optional<T></i> против <i>OptionalInt</i>	662
244. Подтверждение эквивалентности объектов <i>Optional</i>	662
245. Преобразование значений посредством методов <i>map()</i> и <i>flatMap()</i>	663
246. Фильтрация значений посредством метода <i>Optional.filter()</i>	665
247. Выстраивание API <i>Optional</i> и API потоков в цепочку.....	666
248. Класс <i>Optional</i> и операции, чувствительные к идентичности.....	667
249. Возвращение значения типа <i>boolean</i> при пустом объекте класса <i>Optional</i>	668
Резюме.....	669

Глава 13. API HTTP-клиента и протокола WebSocket	670
Задачи	670
Решения.....	672
250. Протокол HTTP/2.....	672
251. Инициирование асинхронного запроса <i>GET</i>	673
Построитель параметров запроса	675
252. Настройка прокси-селектора	675
253. Настройка/получение заголовков.....	676
Настройка заголовков запросов.....	676
Получение заголовков запроса/отклика.....	677
254. Настройка метода HTTP	678
255. Настройка тела запроса.....	679
Создание тела из строки текста	679
Создание тела из потока <i>InputStream</i>	680
Создание тела из массива байтов	680
Создание тела из файла	681
256. Настройка аутентификации соединения	681
257. Настройка таймаута.....	682
258. Настройка политики перенаправления.....	683
259. Отправка синхронных и асинхронных запросов	684
Отправка запроса синхронно	684
Отправка запроса асинхронно	684
Одновременная отправка многочисленных запросов.....	685
260. Обработка файлов cookie	686
261. Получение информации отклика	687
262. Обработка тела отклика	687
Обработка тела отклика как строки	688
Обработка тела отклика как файла.....	688
Обработка тела отклика как массива байтов.....	688
Обработка тела отклика как входного потока.....	688
Обработка тела отклика как потока строк.....	689
263. Получение, обновление и сохранение данных JSON.....	689
JSON-отклик в объект <i>User</i>	691
Обновленный объект <i>User</i> в JSON-запрос.....	691
Новый объект <i>User</i> в JSON-запрос.....	692
264. Сжатие	692
265. Обработка данных формы	693
266. Скачивание ресурса с сервера	695
267. Закачивание ресурса на сервер многочастным файлом.....	696
268. Серверная push-отправка по протоколу HTTP/2	698
269. Протокол WebSocket	702
Резюме.....	704
Предметный указатель.....	705

Составители

Об авторе

Анджел Леонард (Anghel Leonard) — главный технологический стратег с более чем 20-летним опытом работы в экосистеме Java. В своей повседневной работе он сосредоточен на архитектурном планировании и разработке распределенных приложений Java, которые обеспечивают робастные архитектуры, чистый код и высокую производительность. Он также увлечен консультированием, наставничеством и техническим руководством.

Он автор нескольких книг, видеороликов и десятков статей, связанных с технологиями Java.

О рецензентах

Кристиан Станкалау (Cristian Stancalau) имеет степень магистра и бакалавра в области компьютерных наук и инженерии Университета Бейба-Бойаи (Румыния), где он с 2018 года работает ассистентом лектора. В настоящее время он занимает должность главного архитектора программно-информационного обеспечения и специализируется на ревизии корпоративного кода в DevFactory.

Ранее он был соучредителем стартапа в области видеотехнологий и руководил им в качестве технического директора. Кристиан имеет опыт наставничества и преподавания как в коммерческом, так и в академическом секторах, занимаясь консультированием по технологиям Java и архитектуре продуктов.

"Хотел бы поблагодарить Анджела Леонарда за честь поручить мне выполнить технический обзор книги „Задачи кодирования в среде Java“. Чтение этой книги было для меня настоящим удовольствием, и я уверен, что так и будет и для ее читателей".

Вишну Говиндрао Кулкарни (Vishnu Govindrao Kulkarni) является увлеченным фрилансером и поставщиком решений (в сотрудничестве с Fortune Consulting). У него обширный опыт работы в различных областях, восьмилетний опыт работы с полным стеком Java, Java Spring, Spring Boot, API Hibernate REST и Oracle. Он сотрудничал с несколькими организациями в области строительства корпоративных решений с использованием среды Java и вычислительных каркасов Java. В настоящее время Вишну продолжает конструировать и разрабатывать решения, тесно взаимодействуя с клиентами, помогая им в извлечении выгоды из этих решений.

Ранее он работал техническим рецензентом книги "Основы Java" (Java Fundamentals) для издательства Packt.

Предисловие

Сверхбыстрая эволюция JDK между версиями 8 и 12 удлинила кривую усвоения современной среды Java и, следовательно, увеличила время, необходимое для выведения разработчиков на плато продуктивности. Ее новые средства и понятия можно использовать для решения различных современных задач. Настоящая книга позволит вам принять на вооружение объективный подход к часто встречающимся задачам через объяснение правильных практических приемов и технических решений в отношении сложности, производительности, удобочитаемости и многого другого.

Книга поможет вам выполнять вашу ежедневную работу и укладываться в сроки. Здесь вы найдете 300+ приложений, содержащих 1000+ примеров, охватывающих часто встречающиеся и основополагающие области: строки, числа, массивы, коллекции, структуры данных, даты и время, немутуируемость, логический вывод типов, класс `Optional`, ввод/вывод в Java, рефлексия в Java, функциональное программирование, конкурентность и API HTTP-клиента. Прокачайте свои навыки с помощью задач, которые были тщательно разработаны, чтобы высветить и охватить ключевые знания, к которым приходится обращаться в повседневной работе. Другими словами (независимо от того, является ли ваша работа легкой, средней или сложной), наличие этих знаний в вашем арсенале является обязательным и не сравнимым ни с чем.

К концу чтения этой книги вы получите глубокое понимание понятий среды Java и обретете уверенность в разработке и выборе правильных решений ваших задач.

Для кого эта книга предназначена

Книга будет особенно полезной для разработчиков в среде Java начального и промежуточного уровней. Однако рассмотренные здесь задачи встречаются в повседневной работе любого разработчика в среде Java.

Необходимая техническая предподготовка довольно проста. Вы главным образом должны быть поклонником среды Java и иметь хорошие навыки и интуицию в искусстве следования за фрагментом кода на языке Java.

Что эта книга охватывает

Глава 1 "Строки, числа и математика" содержит 39 задач с привлечением строк, чисел и математических операций. Она начинается с ряда классических задач для строк, таких как подсчет повторов, инвертирование строки и удаление пробелов. Затем в ней рассматриваются задачи, посвященные числам и математическим операциям, таким как сложение двух больших чисел, переполнение операции, сравнение двух беззнаковых чисел, вычисление целой части и остатка от деления и многое другое. Каждая задача проводится через несколько решений, включая функциональный стиль Java 8. Более того, эта глава охватывает задачи, которые можно решить в JDK 9, 10, 11 и 12 за счет интерфейсов `Future`.

Глава 2 "Объекты, немутуируемость и выражения `switch`" содержит 18 задач с привлечением объектов, немутуируемости и выражений `switch`. Эта глава начинается с нескольких задач, касающихся работы со ссылками `null`. Она продолжается задачами, касающимися проверки индексов, методов `equals()` и `hashCode()`, а также немутуируемости (например, написание немутуируемых классов и передача/возврат мутуируемых объектов из немутуируемых классов). Последняя часть главы посвящена клонированию объектов и выражениям `switch` в версии JDK 12.

Глава 3 "Работа с датой и временем" содержит 20 задач с привлечением даты и времени. Их цель — охватить широкий круг тем (конвертирование, форматирование, сложение, вычитание, определение периодов/длительностей, вычисление даты и времени и т. д.) посредством классов `Date`, `Calendar`, `LocalDate`, `LocalTime`, `LocalDateTime`, `ZoneDateTime`, `OffsetDateTime`, `OffsetTime`, `Instant` и т. д. К концу этой главы у вас не будет проблем с формированием даты и времени, отвечающим потребностям вашего приложения.

Глава 4 "Логический вывод типов" содержит 21 задачу с привлечением JEP 286, или логического вывода типа локальной переменной Java (`Local Variable Type Inference LVTI`), так называемого типа `var`. Решения были тщательно проработаны, раскрывая лучшие практические приемы и часто встречающиеся ошибки, связанные с использованием типа `var`. К концу этой главы вы будете знать о `var` всё, что нужно, для портирования его в производственные приложения.

Глава 5 "Массивы, коллекции и структуры данных" содержит 30 задач с привлечением массивов, коллекций и нескольких структур данных. Ее цель состоит в том, чтобы обеспечить решения категорий задач, возникающих в широком спектре приложений, таких как сортировка, поиск, сравнение, упорядочивание, инвертирование, заполнение, слияние, копирование, замена и т. д. Представленные решения имплементированы в Java 8–12 и могут быть использованы в качестве основы для решения других родственных задач. К концу этой главы у вас будет солидная база знаний, которая пригодится для решения разнообразных задач с привлечением массивов, коллекций и структур данных.

Глава 6 "Пути, файлы, буферы, сканирование и форматирование ввода/вывода в среде Java" содержит 20 задач с привлечением ввода/вывода файлов в Java. От манипулирования, перемещения и просмотра путей до потоковой передачи файлов и эффективных способов чтения/записи текстовых и двоичных файлов, мы рассмотрим задачи, которые являются обязательными в арсенале любого разработчика в среде Java. Благодаря навыкам, полученным из этой главы, вы сможете решать большинство часто встречающихся задач, связанных с файловым вводом/выводом Java.

Глава 7 "Классы, интерфейсы, конструкторы, методы и поля в API рефлексии Java" содержит 17 задач, связанных с API рефлексии Java. От классических тем, таких как проверка и создание экземпляров артефактов Java (например, модулей, пакетов, классов, интерфейсов, надклассов, конструкторов, методов, аннотаций, массивов и т. д.), до синтетических и мостовых конструкций или управления гнездовым доступом (JDK 11), в этой главе представлен полный обзор API рефлексии Java.

Глава 8 "Программирование в функциональном стиле — основы и шаблоны архитектурного дизайна" содержит 11 задач с привлечением программирования в среде Java в функциональном стиле. Данная глава начинается с задачи, которая призвана обеспечить исчерпывающую информацию о переходе от кода без интерфейсов к коду с функциональными интерфейсами. Она продолжается набором шаблонов архитектурного дизайна из книги "банды четырех"¹, проинтерпретированных в функциональном стиле Java.

Глава 9 "Программирование в функциональном стиле — глубокое погружение" содержит 22 задачи с привлечением функционального стиля программирования в среде Java. Здесь мы сосредоточимся на нескольких проблемах, связанных с классическими операциями, которые встречаются в потоках (например, `filter` и `map`), и обсудим бесконечные потоки, `null`-безопасные потоки и методы по умолчанию. Полный список задач охватывает группирование, разбиение и коллекторы, включая коллектор JDK 12 `teeing()` и написание собственного коллектора. В добавление к этому рассматриваются функции `takeWhile()`, `dropWhile()`, композиция функций, предикатов и компараторов, тестирование и отладка лямбда-выражений и другие животрепещущие темы.

Глава 10 "Конкурентность — пулы нитей исполнения, объекты `Callable` и синхронизаторы" содержит 14 задач с привлечением конкурентности (псевдоодновремен-

¹ Книга "Шаблоны архитектурного дизайна: элементы многократно-используемого программно-информационного обеспечения" Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона и Джона Влиссайда (Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. — Addison-Wesley Professional, 1994), в просторечии именуемая книгой "банды четырех", представляет собой каталог шаблонов объектно-ориентированного конструирования программно-информационного обеспечения. — *Прим. перев.*

ной обработки) в Java. Эта глава начинается с нескольких основополагающих задач, связанных с жизненными циклами нитей исполнения и применением замков на уровне объектов и классов. Она продолжается рядом задач, касающихся пулов нитей исполнения в среде Java, включая пул обкрадывающих нитей исполнения JDK 8. После этого мы займемся задачами, посвященными интерфейсам `Callable` и `Future`. Мы также посвятим несколько задач синхронизаторам Java (в частности, барьеру, семафору и обменнику). Прочитав эту главу, вы будете знать основные понятия одновременной обработки в Java и будете готовы продолжить работу с целым рядом продвинутых задач.

Глава 11 "Конкурентность — глубокое погружение" содержит 13 задач с привлечением конкурентности (псевдоодновременной обработки) в Java. Эта глава охватывает такие области, как каркас разветвления/соединения, классы `CompletableFuture`, `ReentrantLock`, `ReentrantReadWriteLock`, `StampedLock`, атомарные переменные, отмена операций, прерываемые методы, переменные типа `ThreadLocal` и тупики. Закончив чтение этой главы, вы получите значительный объем знаний о конкурентности, необходимых любому разработчику в среде Java.

Глава 12 "Класс `Optional`" содержит 24 задачи, призванные привлечь ваше внимание к нескольким правилам работы с классом `Optional`. Задачи и решения, представленные в этой главе, основаны на определении Брайана Гетца (архитектора языка Java): "Класс `Optional` предназначен для обеспечения ограниченного механизма для типов, возвращаемых из библиотечных методов, где должен иметься ясный способ представления результата, и использование `null` для таких методов с подавляющей вероятностью будет становиться причиной ошибок". Но там, где есть правила, есть и исключения. Поэтому не следует делать вывод о том, что правила (или практические решения), представленные здесь, должны соблюдаться (или избегаться) любой ценой. Как всегда, решение зависит от задачи.

Глава 13 "API HTTP-клиента и протокола `WebSocket`" содержит 20 задач, цель которых — охватить API HTTP-клиента и API протокола `WebSocket`. Помните, был такой подкласс `URLConnection`? Так вот, JDK 11 поставляется с API HTTP-клиента как переосмыслением подкласса `URLConnection`. API HTTP-клиента является простым в использовании и поддерживает протоколы HTTP/2 (по умолчанию) и HTTP/1.1. Для обеспечения обратной совместимости API HTTP-клиента переходит с HTTP/2 на HTTP/1.1 автоматически, если сервер не поддерживает HTTP/2. Более того, API HTTP-клиента поддерживает синхронные и асинхронные модели программирования и опирается на потоки для передачи данных (реактивные потоки). Он также поддерживает протокол `WebSocket`, который используется в веб-приложениях реального времени с целью обеспечения связи между клиентом и сервером с низкими накладными расходами на пересылку сообщений.

Как получить максимальную отдачу от этой книги

Вы должны иметь базовые знания языка и среды Java. Вам также следует установить:

- ◆ IDE (рекомендуется, но не обязательно, иметь Apache NetBeans 11.x: <https://netbeans.apache.org/>);
- ◆ JDK 12 и Maven 3.3.x;
- ◆ дополнительные сторонние библиотеки, которые необходимо будет установить в нужный момент (ничего особенного).

Файлы с примерами исходного кода

Файлы с примерами можно скачать с вашего аккаунта по адресу <http://www.packtpub.com/> для всех книг издательства Packt Publishing, которые вы приобрели. Если вы купили эту книгу в другом месте, то можно посетить <http://www.packtpub.com/support> и зарегистрироваться там, чтобы получить файлы прямо по электронной почте.

Скачать файлы с примерами можно, выполнив следующие шаги:

1. Войдите на наш веб-сайт или зарегистрируйтесь там, используя ваш адрес электронной почты и пароль.
2. Наведите указатель мыши на вкладку **SUPPORT** вверху страницы.
3. Щелкните по разделу **Code Downloads & Errata**, посвященному примерам программного кода и опечаткам.
4. Введите название книги в поле поиска.

Скачав файл, убедитесь, что вы распаковали или извлекли папку, воспользовавшись последней версией указанных ниже архиваторов:

- ◆ WinRAR/7-Zip для Windows;
- ◆ Zipeg/iZip/UnRarX для Mac OS;
- ◆ 7-Zip/PeaZip для Linux.

Помимо этого, комплект примеров программного кода, прилагаемый к данной книге, размещен на GitHub в разделе Packt по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>. В случае обновления программного кода он будет обновлен в существующем репозитории GitHub.

Мы также располагаем другими комплектами примеров программного кода, которые можно выбрать из нашего богатого каталога книг и видеороликов, предлагаемого на странице <https://github.com/PacktPublishing/>. Можете убедиться сами!

Цветные изображения

Мы также предоставляем PDF-файл с цветными изображениями скриншотов/схем, используемых в этой книге. Вы можете скачать его отсюда: https://static.packt-cdn.com/downloads/9781789801415_ColorImages.pdf².

Код в действии

Для того чтобы увидеть процесс исполнения кода, перейдите по ссылке: <http://bit.ly/2kSgFKf>.

Условные обозначения

В данной книге используется ряд условных обозначений.

КодВТексте обозначает кодовые слова в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, имена путей, фиктивные URL-адреса, ввод данных пользователем и дескрипторы Twitter. Вот пример: "если текущий символ существует в экземпляре отображения Map, то просто увеличьте число его появлений на 1, используя (символ, появления+1)".

Блок кода задается следующим образом:

```
public Map<Character, Integer> countDuplicateCharacters(String str) {
    Map<Character, Integer> result = new HashMap<>();

    // либо используйте for(char ch: str.toCharArray()) { ... }
    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);

        result.compute(ch, (k, v) -> (v == null) ? 1 : ++v);
    }

    return result;
}
```

Когда нужно привлечь ваше внимание к определенной части фрагмента кода, соответствующие строки или элементы выделяются **жирным шрифтом**:

```
for (int i = 0; i < str.length(); i++) {
    int cp = str.codePointAt(i);
    String ch = String.valueOf(Character.toChars(cp));
    if(Character.charCount(cp) == 2) { // 2 означает суррогатную пару
        i++;
    }
}
```

² Адаптированные цветные изображения для русского издания книги можно скачать со страницы книги на веб-сайте издательства www.bhv.ru. — Прим. ред.

Любой ввод или вывод командной строки записывается следующим образом:

```
$ mkdir css
```

```
$ cd css
```

Жирный шрифт обозначает новый термин, важное слово или слова, которые вы видите на экране, фактические URL-адреса. Например, элементы интерфейса из меню или диалоговых окон в тексте выделены именно полужирным шрифтом. Вот пример: "В Java логический оператор **AND** представлен в виде &&, логический оператор **OR** представлен в виде ||, а логический оператор **XOR** представлен в виде ^".



Данный элемент обозначает подсказку или совет.



Данный элемент обозначает предупреждение или предостережение.

Комментарии переводчика

С целью приведения терминологии в соответствие с общепринятой мировой практикой в настоящем переводе использован ряд уточненных терминов, перечисленных ниже.

Термин implementation. В информатике проводится четкое различие в фазах инженерии программно-информационного обеспечения: архитектурный дизайн, имплементация и реализация. Под *имплементацией* подразумевается воплощение архитектурного дизайна в структурах данных, классах, объектах, функциях, интерфейсах и т. п. Под *реализацией* понимается воплощение в готовом инструменте или законченном продукте. В переводе за основу приняты транслитерация английского термина "имплементировать" и производные.

Термины concurrent, thread и lock. Многозадачная операционная система способна обрабатывать сразу несколько процессов, при этом процесс может состоять из нескольких нитей³ исполнения (thread), где нить исполнения представляет собой главную инфраструктурную единицу планирования в операционной системе.

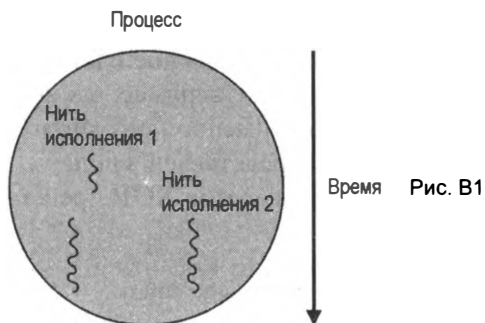
Каждая нить выполняет собственную операцию (псевдо)одновременно. Приставка "псевдо" означает, что "одновременность" достигается за счет того, что процессор переключается между нитями при помощи планировщика, порождая, запуская, приостанавливая и завершая нити исполнения (см. рис. B1⁴).

Нить исполнения может обращаться к совместному ресурсу, причем она делает это конкурентно (concurrently — псевдоодновременно) с использованием замка (lock),

³ Нить — то же самое, что и поток. — *Прим. ред.*

⁴ Рисунок взят из Википедии по адресу [https://simple.wikipedia.org/wiki/Thread_\(computer_science\)](https://simple.wikipedia.org/wiki/Thread_(computer_science)). — *Прим. перев.*

временно блокирующего доступ к совместному ресурсу, пока тот занят. В этом нить отличается от процесса, который так не делает⁵.



Процессы могут обрабатываться операционной системой параллельно; это достигается за счет применения нескольких процессоров/ядер и ресурсов ввода/вывода. Таким образом, конкурентная обработка возможна на одном процессоре с использованием многочисленных нитей исполнения, тогда как параллельная обработка возможна только на двух и более процессорах/ядрах.

Термин *thread* этимологически восходит к протогерманскому предку, который по форме и смыслу означает "нить" и является родственником нашим "прясти", "прядь", т. е. процесс сплетается из нитей (ср. нить рассуждения, нить дискуссии; spawning a new thread for each stream — порождение новой нити исполнения для каждого потока данных).

Термин *concurrent* в зависимости от контекста означает "одновременный, совпадающий, сосуществующий" и складывается из приставки *con* (вместе) и основы *occurrence* (появление). К примеру, в лингвистике существует термин *co-occurrence* (коокуренность) и его разновидность *concurrence*, которыми обозначается частота появления в одном месте двух или более слов в словарном корпусе.

Термин "lock" означает "замок", "замковый механизм". В информатике замок — это механизм контроля доступа к данным с целью их защиты. В программировании замки часто используются для того, чтобы несколько программ или нитей исполнения могли использовать ресурс совместно — например, обращаться к файлу для его обновления — на поочередной основе⁶. Иными словами, замки являются механизмом синхронизации, абстракцией, которая позволяет не более чем одной нити владеть им одновременно. Владение замком — это состояние, при котором одна нить как бы сообщает другим нитям: "Я модифицирую этот элемент, прямо сейчас не трогайте его". Приобретение замка позволяет нити иметь эксклюзивный доступ к совместным данным, охраняемый этим замком, заставляя другие нити блокировать свою работу.

⁵ Более подробное описание нитей исполнения см. в статье на Хабре "Что такое нити (threads)" (<https://habr.com/ru/post/40275/>). — Прим. перев.

⁶ См. <https://search400.techtarget.com/definition/lock> и https://www.webopedia.com/TERM/R/resource_locking.html. — Прим. перев.

1

Строки, числа и математика

Эта глава включает 39 задач с привлечением строк, чисел и математических операций. Мы начнем с рассмотрения ряда классических задач для строковых значений, таких как подсчет повторов, инвертирование строки и удаление пробелов. Затем мы рассмотрим задачи, посвященные числам и математическим операциям, таким как сложение двух больших чисел и переполнение операции, сравнение двух беззнаковых чисел, вычисление целой части деления и целой части остатка от деления. Каждая задача проводится через несколько решений, включая функциональный стиль Java 8. Более того, мы осветим задачи, которые касаются JDK 9, 10, 11 и 12.

К концу этой главы вы должны будете знать, как использовать целый ряд технических приемов, позволяющих вам манипулировать строковыми значениями, а также применять, корректировать и адаптировать их ко многим другим задачам. Вы будете знать, как решать крайние математические случаи, которые могут приводить к странным и непредсказуемым результатам.

Задачи

Используйте следующие задачи для проверки вашего умения программировать строки и крайние математические случаи. Перед тем как обратиться к решениям и скачать примеры программ, настоятельно рекомендуется изучить каждую задачу.

1. **Подсчет повторяющихся символов.** Написать программу, которая подсчитывает повторяющиеся символы в заданной строке.
2. **Отыскание первого неповторяющегося символа.** Написать программу, которая возвращает первый неповторяющийся (некратный) символ в заданной строке.
3. **Инвертирование букв и слов.** Написать программу, которая инвертирует буквы каждого слова, и программу, которая инвертирует буквы каждого слова и сами слова.
4. **Проверка, содержит ли строковое значение только цифры.** Написать программу, которая проверяет, что заданная строка содержит только цифры.

5. **Подсчет гласных и согласных.** Написать программу, которая подсчитывает число гласных и согласных букв в заданной строке. Сделать это для английского языка, который имеет пять гласных (*a, e, i, o и u*).
6. **Подсчет появлений некоторого символа.** Написать программу, которая подсчитывает появления того или иного символа в заданной строке.
7. **Конвертирование строки в значение типа `int`, `long`, `float` или `double`.** Написать программу, которая конвертирует заданный объект типа `String` (представляющий число) в значение типа `int`, `long`, `float` или `double`.
8. **Удаление пробелов из строки.** Написать программу, которая удаляет все пробелы из заданной строки.
9. **Соединение нескольких строк с помощью разделителя.** Написать программу, которая соединяет заданные строки с помощью заданного разделителя.
10. **Генерирование всех перестановок.** Написать программу, которая генерирует все перестановки заданной строки.
11. **Проверка, что строка является палиндромом.** Написать программу, которая выясняет, является ли данная строка палиндромом или нет.
12. **Удаление повторяющихся символов.** Написать программу, которая удаляет повторяющиеся символы из заданной строки.
13. **Удаление заданного символа.** Написать программу, которая удаляет заданный символ из заданной строки.
14. **Отыскание символа с наибольшим числом появлений.** Написать программу, которая отыскивает символ с наибольшим числом появлений в заданной строке.
15. **Сортировка массива строк по длине.** Написать программу, которая сортирует заданный массив по длине строк.
16. **Проверка наличия подстроки в строке.** Написать программу, которая проверяет, содержит ли заданная строка заданную подстроку.
17. **Подсчет числа появлений подстроки в строке.** Написать программу, которая подсчитывает число появлений заданной строки в другой заданной строке.
18. **Проверка, являются ли две строки анаграммами.** Написать программу, которая проверяет, являются ли две строки анаграммами. Учесть, что анаграммой строки является строка, образованная перестановкой букв в обратном порядке с игнорированием заглавных букв и пробелов.
19. **Объявление многострочных строковых литералов (или текстовых блоков).** Написать программу, которая объявляет многострочные строковые литералы или текстовые блоки.
20. **Конкатенирование одной и той же строки n раз.** Написать программу, которая конкатенирует одну и ту же строку заданное число раз.

21. **Удаление начальных и замыкающих пробелов.** Написать программу, которая удаляет начальные и замыкающие пробелы заданной строки.
22. **Отыскание наибольшего общего префикса.** Написать программу, которая отыскивает наибольший общий префикс заданных строк.
23. **Применение отступа.** Написать несколько фрагментов кода с применением отступа к заданному тексту.
24. **Трансформирование строк.** Написать несколько фрагментов кода с преобразованием строки в другую строку.
25. **Вычисление минимума и максимума двух чисел.** Написать программу, которая возвращает минимум и максимум двух чисел.
26. **Сложение двух крупных чисел типа `int/long` и переполнение операции.** Написать программу, которая складывает два крупных числа `int/long` и выбрасывает арифметическое исключение в случае переполнения операции.
27. **Строка как беззнаковое число с основанием системы счисления.** Написать программу, которая разбирает и конвертирует заданную строку в беззнаковое число (типа `int` или `long`) с заданным основанием системы счисления.
28. **Конвертирование в число посредством беззнаковой конверсии.** Написать программу, которая конвертирует заданное число типа `int` в число типа `long` с помощью беззнаковой конверсии.
29. **Сравнение двух беззнаковых чисел.** Написать программу, которая сравнивает два заданных числа как беззнаковые.
30. **Вычисление частного и остатка от деления беззнаковых значений.** Написать программу, которая вычисляет частное и остаток заданного беззнакового значения.
31. **Значение типа `double/float` является конечным значением с плавающей точкой.** Написать программу, которая определяет, что заданное значение типа `double/float` является конечным значением с плавающей точкой.
32. **Применение логического И/ИЛИ/исключающего ИЛИ к двум булевым выражениям.** Написать программу, которая применяет логическое И/ИЛИ/исключающее ИЛИ к двум булевым выражениям.
33. **Конвертирование значения типа `BigInteger` в примитивный тип.** Написать программу, которая извлекает значение примитивного типа из заданного значения типа `BigInteger`.
34. **Конвертирование значения типа `long` в значение типа `int`.** Написать программу, которая конвертирует значение типа `long` в значение типа `int`.
35. **Вычисление целой части деления и целой части остатка от деления.** Написать программу, которая вычисляет целую часть деления и целую часть остатка от деления делимого (x) на делитель (y).

36. **Следующее значение с плавающей точкой.** Написать программу, которая возвращает следующую плавающую точку, смежную с заданным значением типа `float/double` в направлении положительной и отрицательной бесконечности.
37. **Умножение двух крупных значений типа `int/long` и переполнение операции.** Написать программу, которая умножает два крупных значения типа `int/long` и выбрасывает арифметическое исключение в случае переполнения операции.
38. **Совмещенное умножение-сложение (FMA).** Написать программу, которая берет три значения типа `float/double` (`a`, `b`, `c`) и вычисляет `a*b+c` эффективным способом.
39. **Компактное форматирование чисел.** Написать программу, которая форматирует число 1,000,000¹ в 1М (локаль США) и в 1 mln (итальянская локаль). В добавление к этому, выполнить разбор 1М и 1 mln из строки с конвертацией в число.

Решения

В следующих разделах описаны решения перечисленных выше задач. Помните, что обычно не существует одного единственно правильного варианта решения той или иной задачи. Кроме того, имейте в виду, что приведенные здесь объяснения включают только самые интересные и важные детали, необходимые для решения. Вы можете скачать примеры решений, чтобы увидеть дополнительные детали и поэкспериментировать с программами, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

1. Подсчет повторяющихся символов

Решение задачи подсчета символов в строке (включая специальные символы, такие как `#`, `$` и `%`) предусматривает взятие каждого символа и сравнение его с остальными. Во время сравнения поддерживается состояние счета с помощью числового счетчика, который увеличивается на единицу всякий раз, когда найден текущий символ.

У этой задачи существует два варианта решения.

Первый вариант решения перебирает строковые символы и использует специальную структуру — отображение `Map`² — для хранения символов в качестве ключей и

¹ Здесь оставлена форма записи чисел, принятая в оригинале книги, где для отделения дробной части принято использовать точку, и для отделения групп по три числа многозначного числа принято использовать запятую. — *Прим. перев.*

² Отображение (`map`) — это модель функции в математическом смысле, как отношения, которое уникально ассоциирует члены одного множества с членами другого множества в виде перечня пар "ключ–значение" с доступом к значению по ключу. В других языках часто называется словарем, ассоциативным массивом или таблицей "ключ–значение". — *Прим. перев.*

числа появлений в качестве значений. Если текущий символ ни разу не был добавлен в отображение `Map`, то добавить его как `(символ, 1)`. Если текущий символ в отображении `Map` существует, то просто увеличить число его появлений на 1, например `(символ, появления+1)`. Это показано в следующем ниже фрагменте кода:

```
public Map<Character, Integer> countDuplicateCharacters(String str) {
    Map<Character, Integer> result = new HashMap<>();

    // либо использовать for(char ch: str.toCharArray()) { ... }
    for (int i = 0; i<str.length(); i++) {
        char ch = str.charAt(i);

        result.compute(ch, (k, v) -> (v == null) ? 1 : ++v);
    }

    return result;
}
```

Еще один вариант решения основан на потоковом функционале Java 8. Это решение состоит из трех шагов. Первые два шага предусматривают трансформирование заданной строки в `Stream<Character>`, тогда как последний шаг отвечает за группирование и подсчет символов. Вот эти шаги:

1. Вызвать метод `String.chars()` на исходной строке. В результате будет возвращен экземпляр класса `IntStream`. Поток `IntStream` содержит целочисленное представление символов в заданной строке.
2. Трансформировать экземпляр класса `IntStream` в поток символов посредством метода `mapToObj()` (конвертировать целочисленное представление в дружественную для человека символьную форму).
3. Наконец, сгруппировать символы (`Collectors.groupingBy()`) и подсчитать их (`Collectors.counting()`).

Следующий ниже фрагмент кода склеивает эти три шага в один метод:

```
public Map<Character, Long> countDuplicateCharacters(String str) {
    Map<Character, Long> result = str.chars()
        .mapToObj(c -> (char) c)
        .collect(Collectors.groupingBy(c -> c, Collectors.counting()));

    return result;
}
```

О символах Юникода

Нам довольно хорошо знакомы символы ASCII: у нас есть непечатные управляющие коды между 0–31, печатные символы между 32–127 и расширенные коды ASCII между 128–255. Но как насчет символов Юникода? Ниже приведен целый

раздел, который посвящен каждой задаче, требующей манипулирования символами Юникода.

Итак, в двух словах, ранние версии Юникода содержали символы со значениями менее 65 535 (0xFFFF). Java представляет эти символы, используя 16-битный тип `char`. Вызов метода `charAt(i)` работает, как и ожидалось, до тех пор, пока `i` не превысит 65 535. Но со временем Юникод добавил еще больше символов, и максимальное значение достигло 1 114 111 (0x10FFFF). Эти символы не укладываются в 16 бит, и поэтому были предусмотрены 32-битные значения (так называемые *код-овые точки*) для схемы кодирования UTF-32.

Но, к сожалению, Java не поддерживает UTF-32! И тем не менее стандарт Юникода нашел решение с использованием тех же 16 бит для представления таких символов. Указанное решение предусматривает следующее:

♦ 16-битные *высокие суррогаты*: 1024 значения (от U+D800 до U+DBFF);

♦ 16-битные *низкие суррогаты*: 1024 значения (от U+DC00 до U+DFFF).

Высокий суррогат с последующим низким суррогатом определяет *суррогатную пару*. Суррогатные пары используются для представления значений между 65 536 (0x10000) и 1 114 111 (0x10FFFF). Таким образом, некоторые символы, известные как дополнительные символы Юникода, представляются в виде суррогатных пар Юникода (один символ, или знак, помещается в пространстве пары символов), которые объединяются в одну кодовую точку. Среда Java использует преимущества этого представления и обеспечивает его с помощью набора таких методов, как `codePointAt()`, `codePoints()`, `codePointCount()` и `offsetByCodePoints()` (подробности см. в документации Java). Вызов метода `codePointAt()` вместо метода `charAt()`, метода `codePoints()` вместо метода `chars()` и т. д. помогает нам писать решения, которые также охватывают символы ASCII и Юникода.

Например, хорошо известный символ двух сердец — это суррогатная пара Юникода, которая представляется как массив `char[]`, содержащий два значения: `\uD83D` и `\uDC95`. Кодовая точка этого символа равна 128149. Для того чтобы из этой кодовой точки получить значение типа `String`, надо вызвать `String str = String.valueOf(Character.toChars(128149))`. Подсчитать кодовые точки в переменной `str` можно путем вызова метода `str.codePointCount(0, str.length())`, который возвращает 1, даже если длина `str` равна 2. Вызов `str.codePointAt(0)` возвращает 128149, а вызов `str.codePointAt(1)` возвращает 56469. Вызов метода `Character.toChars(128149)` возвращает 2, поскольку для представления этой кодовой точки, являющейся суррогатной парой Юникода, требуется именно два символа. Для 16-битных символов ASCII и Юникода он будет возвращать 1.

Таким образом, если мы попытаемся переписать первый вариант решения (который перебирает символы строкового значения и использует отображение `Map` для хранения символов в виде ключей и числа появлений в виде значений) с целью поддерж-

ки ASCII и Юникода (включая суррогатные пары), мы получим следующий ниже исходный код:

```
public static Map<String, Integer> countDuplicateCharacters(String str) {
    Map<String, Integer> result = new HashMap<>();

    for (int i = 0; i < str.length(); i++) {
        int cp = str.codePointAt(i);
        String ch = String.valueOf(Character.toChars(cp));
        if (Character.charCount(cp) == 2) { // 2 означает суррогатную пару
            i++;
        }

        result.compute(ch, (k, v) -> (v == null) ? 1 : ++v);
    }

    return result;
}
```

Выделенный жирным шрифтом исходный код можно записать следующим образом:

```
String ch = String.valueOf(Character.toChars(str.codePointAt(i)));
if (i < str.length() - 1 && str.codePointCount(i, i + 2) == 1) {
    i++;
}
```

Наконец, приведенный выше вариант решения можно переписать в функциональном стиле Java 8 с охватом суррогатных пар Юникода следующим образом:

```
public static Map<String, Long> countDuplicateCharacters(String str) {
    Map<String, Long> result = str.codePoints()
        .mapToObj(c -> String.valueOf(Character.toChars(c)))
        .collect(Collectors.groupingBy(c -> c, Collectors.counting()));

    return result;
}
```



Относительно поддержки сторонними библиотеками, пожалуйста, обратитесь к библиотеке Guava и ее коллекции Multiset<String>.

Некоторые представленные далее задачи дают ответы, которые охватывают ASCII, 16-битный Юникод и суррогатные пары Юникода. Эти ответы были выбраны произвольно, и поэтому, опираясь на показанные выше варианты решения, вы можете легко переписать ответы к задачам, которые не предусматривают необходимого варианта.

2. Отыскание первого неповторяющегося символа

Существуют разные варианты решения задачи отыскания первого неповторяющегося (некратного) символа в строке. В общих чертах задача может быть решена за один обход строки либо за более полный или частичный обходы.

В варианте *с одним обходом* мы заполняем массив, предусмотренный для хранения индексов всех символов, которые появляются в строке ровно один раз. Имея этот массив, просто надо вернуть наименьший индекс, в котором содержится неповторяющийся символ:

```
private static final int EXTENDED_ASCII_CODES = 256;
...
public char firstNonRepeatedCharacter(String str) {
    int[] flags = new int[EXTENDED_ASCII_CODES];

    for (int i = 0; i < flags.length; i++) {
        flags[i] = -1;
    }

    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);
        if (flags[ch] == -1) {
            flags[ch] = i;
        } else {
            flags[ch] = -2;
        }
    }

    int position = Integer.MAX_VALUE;

    for (int i = 0; i < EXTENDED_ASCII_CODES; i++) {
        if (flags[i] >= 0) {
            position = Math.min(position, flags[i]);
        }
    }

    return position == Integer.MAX_VALUE ?
        Character.MIN_VALUE : str.charAt(position);
}
```

Этот вариант решения исходит из того, что каждый символ в строке является частью расширенной таблицы ASCII (256 кодов). Наличие кодов больше 256 требует от нас увеличить размер массива соответствующим образом (<http://www.alansofficespace.com/unicode/unicd99.htm>). Указанный вариант решения будет работать до тех пор, пока размер массива не выйдет за пределы наибольшего значения типа `char`, которым является `Character.MAX_VALUE`, т. е. 65 535.

С другой стороны, `Character.MAX_CODE_POINT` возвращает максимальное значение Юникодовой кодовой точки — 1 114 111. Для того чтобы охватить этот интервал, нам нужна другая имплементация, основанная на методах `codePointAt()` и `codePoints()`.

Благодаря варианту решения с одним обходом, это делается довольно быстро. *Еще один вариант* состоит в циклическом переборе каждого символа строки и подсчете числа появлений. Любое второе появление (дубликат) приводит к прерыванию цикла, переходу к следующему символу и повтору алгоритма. Если достигнут конец строки, то алгоритм возвращает текущий символ как первый неповторяющийся символ. Этот вариант решения имеется в исходном коде, прилагаемом к данной книге.

Еще одно решение, приведенное ниже, основано на связанном хеш-отображении `LinkedHashMap`. Эта коллекция Java поддерживает *порядок вставки* (порядок, в котором ключи вставлялись в коллекцию) и является очень удобным для этого варианта решением. Коллекция `LinkedHashMap` заполняется символами в качестве ключей и числом появлений в качестве значений. После того как коллекция `LinkedHashMap` будет заполнена, будет возвращен первый ключ, значение которого равно 1. Благодаря своей способности поддерживать *порядок вставки* это значение будет возвращать первый неповторяющийся символ в строке:

```
public char firstNonRepeatedCharacter(String str) {
    Map<Character, Integer> chars = new LinkedHashMap<>();

    // либо использовать for(char ch: str.toCharArray()) { ... }
    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);

        chars.compute(ch, (k, v) -> (v == null) ? 1 : ++v);
    }

    for (Map.Entry<Character, Integer> entry: chars.entrySet()) {
        if (entry.getValue() == 1) {
            return entry.getKey();
        }
    }

    return Character.MIN_VALUE;
}
```

В прилагаемом к этой книге исходном коде вышеупомянутое решение было написано в функциональном стиле Java 8. Кроме того, решение в функциональном стиле с поддержкой ASCII, 16-битного Юникода и юникодовых суррогатных пар выглядит следующим образом:

```
public static String firstNonRepeatedCharacter(String str) {
    Map<Integer, Long> chs = str.codePoints()
        .mapToObj(cp -> cp)
```

```
.collect(Collectors.groupingBy(Function.identity(),
    LinkedHashMap::new, Collectors.counting()));

int cp = chs.entrySet().stream()
    .filter(e -> e.getValue() == 1L)
    .findFirst()
    .map(Map.Entry::getKey)
    .orElse(Integer.valueOf(Character.MIN_VALUE));

return String.valueOf(Character.toChars(cp));
}
```

Для того чтобы разобраться в этом коде поглубже, пересмотрите *подразд. "О символах Юникода" в разд. "1. Подсчет повторяющихся символов" ранее в этой главе.*

3. Инвертирование букв и слов

Прежде всего давайте переставим буквы в каждом отдельном слове. Решением этой задачи может быть использование класса `StringBuilder`. Первый шаг состоит в разбиении строки на массив слов с использованием пробела в качестве разделителя (`String.split(" ")`). Далее мы инвертируем каждое слово, используя соответствующие коды ASCII, и добавляем результат в `StringBuilder`. Сначала мы разделяем заданную строку пробелом. Затем обходим полученный массив слов в цикле и переставляем буквы в каждом слове, извлекая каждый символ посредством метода `charAt()` в обратном порядке:

```
private static final String WHITESPACE = " ";

...
public String reverseWords(String str) {
    String[] words = str.split(WHITESPACE);
    StringBuilder reversedString = new StringBuilder();

    for (String word: words) {
        StringBuilder reverseWord = new StringBuilder();

        for (int i = word.length() - 1; i >= 0; i--) {
            reverseWord.append(word.charAt(i));
        }

        reversedString.append(reverseWord).append(WHITESPACE);
    }

    return reversedString.toString();
}
```

Тот же результат можно получить в функциональном стиле Java 8 следующим образом:

```
private static final Pattern PATTERN = Pattern.compile(" +");
...
public static String reverseWords(String str) {
    return PATTERN.splitAsStream(str)
        .map(w -> new StringBuilder(w).reverse())
        .collect(Collectors.joining(" "));
}
```

Обратите внимание, что два предыдущих метода возвращают строку, содержащую буквы каждого слова в обратном порядке, но сами слова находятся в том же исходном порядке.

Теперь давайте рассмотрим еще один метод, который меняет местами элементы каждого слова и сами слова. Благодаря встроенному методу `StringBuilder.reverse()`, это очень легко сделать:

```
public String reverse(String str) {
    return new StringBuilder(str).reverse().toString();
}
```



Относительно поддержки сторонними библиотеками, пожалуйста, обратитесь к библиотеке `Apache Commons Lang` и ее методу `StringUtils.reverse()`.

4. Проверка, содержит ли строковое значение только цифры

Решение этой задачи опирается на методы `Character.isDigit()` и `String.matches()`.

Решение, основанное на методе `Character.isDigit()`, является довольно простым и быстрым — перебрать символы строкового значения и прервать цикл, если этот метод возвращает `false`:

```
public static boolean containsOnlyDigits(String str) {
    for (int i = 0; i < str.length(); i++) {
        if (!Character.isDigit(str.charAt(i))) {
            return false;
        }
    }

    return true;
}
```

Приведенный выше код можно переписать в функциональном стиле Java 8 с помощью метода `anyMatch()`:

```
public static boolean containsOnlyDigits(String str) {
    return !str.chars()
        .anyMatch(n -> !Character.isDigit(n));
}
```

Еще одно решение опирается на метод `String.matches()`. Он возвращает значение типа `boolean`, указывая на то, соответствует ли это строковое значение заданному регулярному выражению:

```
public static boolean containsOnlyDigits(String str) {  
    return str.matches("[0-9]+");  
}
```

Обратите внимание, что решения на основе функционального стиля Java 8 и регулярных выражений обычно являются медленными, поэтому если требуется скорость, лучше опираться на первое решение с использованием метода `Character.isDigit()`.



Следует избегать решение этой задачи посредством методов `parseInt()` или `parseLong()`. Во-первых, отлавливание исключений `NumberFormatException` и выбор дальнейших действий бизнес-логики в блоке `catch` являются плохим практическим приемом. Во-вторых, вместо того чтобы проверять наличие только цифр в строковом значении, эти методы выясняют, является ли строковое значение допустимым числом (например, что строка `-4` является допустимым числом).

Относительно поддержки сторонними библиотеками, пожалуйста, обратитесь к библиотеке `Apache Commons Lang` и ее методу `StringUtils.isNumeric()`.

5. Подсчет гласных и согласных

Следующий далее фрагмент кода предназначен для английского языка, но в зависимости от числа охватываемых вами языков число гласных и согласных может различаться, и код должен быть скорректирован соответствующим образом.

Первое решение этой задачи требует перебора строковых символов в цикле и выполнения следующих ниже действий:

1. Нам нужно проверить, что текущая буква является гласной (это удобно, т. к. в английском языке имеется только пять чистых гласных; в других языках гласных больше, но их число все равно невелико).
2. Если текущая буква не является гласной, то проверить, что она располагается между 'a' и 'z' (это означает, что текущий символ является согласным).

Обратите внимание, что изначально заданный объект `String` переводится в нижний регистр. Это полезно во избежание сравнений с буквами в верхнем регистре. Например, текущий символ сравнивается только с 'a' вместо того, чтобы сравнивать его с 'A' и 'a'.

Исходный код этого решения выглядит следующим образом:

```
private static final Set<Character> allVowels  
    = new HashSet(Arrays.asList('a', 'e', 'i', 'o', 'u'));  
  
public static Pair<Integer, Integer> countVowelsAndConsonants(String str) {  
    str = str.toLowerCase();  
    int vowels = 0;  
    int consonants = 0;
```

```

for (int i = 0; i < str.length(); i++) {
    char ch = str.charAt(i);
    if (allVowels.contains(ch)) {
        vowels++;
    } else if ((ch >= 'a' && ch <= 'z')) {
        consonants++;
    }
}

return Pair.of(vowels, consonants);
}

```

В функциональном стиле Java 8 этот код можно переписать посредством методов `chars()` и `filter()`:

```

private static final Set<Character> allVowels
    = new HashSet(Arrays.asList('a', 'e', 'i', 'o', 'u'));

public static Pair<Long, Long> countVowelsAndConsonants(String str) {
    str = str.toLowerCase();

    long vowels = str.chars()
        .filter(c -> allVowels.contains((char) c))
        .count();

    long consonants = str.chars()
        .filter(c -> !allVowels.contains((char) c))
        .filter(ch -> (ch >= 'a' && ch <= 'z'))
        .count();

    return Pair.of(vowels, consonants);
}

```

Заданная строка фильтруется соответствующим образом, и терминальная операция `count()` возвращает результат. Использование коллектора `partitioningBy()` сожмет код до следующего ниже:

```

Map<Boolean, Long> result = str.chars()
    .mapToObj(c -> (char) c)
    .filter(ch -> (ch >= 'a' && ch <= 'z'))
    .collect(partitioningBy(c -> allVowels.contains(c), counting()));

return Pair.of(result.get(true), result.get(false));

```

Готово! Теперь давайте посмотрим, как подсчитать появления некоторого символа в строке.

6. Подсчет появлений некоторого символа

Простое решение этой задачи состоит из двух следующих шагов:

1. Заменить каждое появление символа в заданном строковом значении на "" (в принципе, это все равно что удалить все появления конкретного символа в заданной строке).
2. Вычесть длину строки, полученную на первом шаге, из длины первоначальной строки.

Исходный код этого метода выглядит следующим образом:

```
public static int countOccurrencesOfACertainCharacter(String str, char ch) {  
    return str.length() - str.replace(String.valueOf(ch), "").length();  
}
```

Следующее ниже решение также охватывает юникодовые суррогатные пары:

```
public static int countOccurrencesOfACertainCharacter(String str, String ch) {  
    if (ch.codePointCount(0, ch.length()) > 1) {  
        // в заданном значении типа String более одного юникодового символа  
        return -1;  
    }  
  
    int result = str.length() - str.replace(ch, "").length();  
  
    // если ch.length() возвращает 2, то это является юникодовой суррогатной парой  
    return ch.length() == 2 ? result / 2 : result;  
}
```

Еще одно простое и быстрое в имплементации решение состоит в переборе строковых символов (в одном цикле) и сравнении каждого символа с заданным символом, при этом нужно увеличивать счетчик на единицу для каждого совпадения:

```
public static int countOccurrencesOfACertainCharacter(String str, char ch) {  
    int count = 0;  
  
    for (int i = 0; i < str.length(); i++) {  
        if (str.charAt(i) == ch) {  
            count++;  
        }  
    }  
  
    return count;  
}
```

Решение, которое охватывает юникодовые суррогатные пары, содержится в исходном коде, прилагаемом к этой книге. В функциональном стиле Java 8 одно из решений состоит в использовании методов `filter()` или `reduce()`.

Например, применение метода `filter()` приведет к следующему фрагменту кода:

```
public static long countOccurrencesOfACertainCharacter(String str, char ch) {
    return str.chars()
        .filter(c -> c == ch)
        .count();
}
```

Решение, которое охватывает юникодовые суррогатные пары, содержится в исходном коде, прилагаемом к этой книге.



TIP

Относительно поддержки сторонними библиотеками, пожалуйста, обратитесь к библиотеке `Apache Commons Lang`, методу `StringUtils.countMatches()`, прикладному каркасу `Spring` и его методу `StringUtils.countOccurrencesOf()` и библиотеке `Guava` и ее методу `CharMatcher.is().countIn()`.

7. Конвертирование строки в значение типа *int*, *long*, *float* или *double*

Рассмотрим следующие ниже строки (можно использовать и отрицательные числа):

```
private static final String TO_INT = "453";
private static final String TO_LONG = "45234223233";
private static final String TO_FLOAT = "45.823F";
private static final String TO_DOUBLE = "13.83423D";
```

Правильное решение задачи конвертирования объекта `String` в значение типа `int`, `long`, `float` или `double` состоит из использования методов классов `Integer`, `Long`, `Float` и `Double` среды `Java` — `parseInt()`, `parseLong()`, `parseFloat()` и `parseDouble()`:

```
int toInt = Integer.parseInt(TO_INT);
long toLong = Long.parseLong(TO_LONG);
float toFloat = Float.parseFloat(TO_FLOAT);
double toDouble = Double.parseDouble(TO_DOUBLE);
```

Конвертировать объект `String` в объекты `Integer`, `Long`, `Float` или `Double` можно посредством методов `Integer.valueOf()`, `Long.valueOf()`, `Float.valueOf()` и `Double.valueOf()`:

```
Integer toInt = Integer.valueOf(TO_INT);
Long toLong = Long.valueOf(TO_LONG);
Float toFloat = Float.valueOf(TO_FLOAT);
Double toDouble = Double.valueOf(TO_DOUBLE);
```

Когда строку не получается конвертировать успешно, язык `Java` выбрасывает исключение `NumberFormatException`. Следующий ниже фрагмент кода говорит сам за себя:

```
private static final String WRONG_NUMBER = "452w";

try {
    Integer toIntWrong1 = Integer.valueOf(WRONG_NUMBER);
} catch (NumberFormatException e) {
```



```
System.err.println(e);
// обработать исключение
}

try {
    int toIntWrong2 = Integer.parseInt(WRONG_NUMBER);
} catch (NumberFormatException e) {
    System.err.println(e);
    // обработать исключение
}
```



Относительно поддержки сторонними библиотеками, пожалуйста, обратитесь к библиотеке `Apache Commons BeanUtils` и ее классам `IntegerConverter`, `LongConverter`, `FloatConverter` и `DoubleConverter`.

8. Удаление пробелов из строки

Решение этой задачи состоит в использовании метода `String.replaceAll()` с регулярным выражением `\s`. В общих чертах, `\s` удаляет все пробелы, включая невидимые, такие как `\t`, `\n` и `\r`:

```
public static String removeWhitespaces(String str) {
    return str.replaceAll("\s", "");
}
```



Начиная с `JDK 11`, `String.isBlank()` проверяет, что строковое значение является пустым или содержит только пробельные кодовые точки. Относительно поддержки сторонними библиотеками, пожалуйста, обратитесь к библиотеке `Apache Commons Lang` и ее методу `StringUtils.deleteWhitespace()` и прикладному каркасу `Spring` и ее методу `StringUtils.trimAllWhitespace()`.

9. Соединение нескольких строк с помощью разделителя

Существует несколько вариантов, идеально подходящих для решения этой задачи. До `Java 8` удобный подход опирался на класс `StringBuilder` следующим образом:

```
public static String joinByDelimiter(char delimiter, String...args) {
    StringBuilder result = new StringBuilder();

    int i = 0;
    for (i = 0; i < args.length - 1; i++) {
        result.append(args[i]).append(delimiter);
    }
    result.append(args[i]);

    return result.toString();
}
```

Начиная с Java 8, существует по крайней мере еще три варианта решения этой задачи. Один из этих вариантов опирается на служебный класс `StringJoiner`, который можно использовать для построения последовательности символов через разделитель (например, запятую).

Он также поддерживает необязательные префикс и суффикс (здесь игнорируются):

```
public static String joinByDelimiter(char delimiter, String...args) {
    StringJoiner joiner = new StringJoiner(String.valueOf(delimiter));

    for (String arg: args) {
        joiner.add(arg);
    }

    return joiner.toString();
}
```

Еще один вариант решения опирается на метод `String.join()`. Этот метод был введен в Java 8 и поставляется в двух разновидностях:

```
String join(CharSequence delimiter, CharSequence... elems)
String join(CharSequence delimiter,
    Iterable<? extends CharSequence> elems)
```

Пример соединения нескольких строк, разделенных пробелом, выглядит следующим образом:

```
String result = String.join(" ", "как", "твои", "дела"); // как твои дела
```

Потоки Java 8 и метод `Collectors.joining()` также будут полезны:

```
public static String joinByDelimiter(char delimiter, String...args) {
    return Arrays.stream(args, 0, args.length)
        .collect(Collectors.joining(String.valueOf(delimiter)));
}
```



Обратите внимание на конкатенацию строк посредством оператора `+=` и методов `concat()` и `String.format()`. Их можно использовать для соединения нескольких строк, но они склонны ухудшать производительность. Например, следующий ниже фрагмент кода опирается на `+=` и работает намного медленнее, чем `StringBuilder`:

```
String str = "";
for(int i = 0; i < 1_000_000; i++) {
    str += "x";
}
```

Оператор `+=` добавляется к строковому значению и реконструирует новое строковое значение, а на это требуется время.

Относительно поддержки сторонними библиотеками, пожалуйста, обратитесь к библиотеке `Apache Commons Lang` и ее методу `StringUtils.join()`, а также к библиотеке `Guava` и ее классу `Joiner`.

10. Генерирование всех перестановок

Решения задач с участием перестановок часто задействуют рекурсивность. В сущности, рекурсивность определяется как процесс, в котором задано некоторое начальное состояние и каждое *последующее состояние* определяется в терминах *предыдущего состояния*.

В нашем случае состояние материализуется буквами заданной строки. Начальное состояние содержит начальную строку, и каждое последующее состояние вычисляется по такой формуле: каждая буква строки будет становиться первой буквой строки (обмен позициями), а затем следует переставить все оставшиеся буквы с помощью рекурсивного вызова. Хотя существуют нерекурсивные или другие рекурсивные варианты решения, указанный вариант решения этой задачи является классическим.

Предложенный вариант решения задачи для строки ABC можно представить так, как показано на рис. 1.1 (обратите внимание на то, как делаются перестановки).



Рис. 1.1

Описанный выше алгоритм в исходном коде выглядит примерно так:

```
public static void permuteAndPrint(String str) {
    permuteAndPrint("", str);
}

private static void permuteAndPrint(String prefix, String str) {
    int n = str.length();

    if (n == 0) {
        System.out.print(prefix + " ");
    } else {
        for (int i = 0; i < n; i++) {
            permuteAndPrint(prefix + str.charAt(i),
                str.substring(i + 1, n) + str.substring(0, i));
        }
    }
}
```

Изначально префикс должен быть пустой строкой (""). На каждой итерации префикс будет конкатенировать (фиксировать) следующую букву строки. Оставшиеся буквы пропускаются через этот метод снова.

Предположим, что этот метод располагается в служебном классе с именем `Strings`. Вы можете вызвать его вот так:

```
Strings.permuteAndStore("ABC");
```

И на выходе вы получите вот такие результаты:

```
ABC ACB BCA BAC CAB CBA
```

Обратите внимание, что этот вариант решения выводит результат на экран. Сохранение результата предусматривает добавление в имплементацию класса `Set`. Предпочтительно использовать именно `Set`, т. к. он устраняет повторы:

```
public static Set<String> permuteAndStore(String str) {
    return permuteAndStore("", str);
}

private static Set<String> permuteAndStore(String prefix, String str) {
    Set<String> permutations = new HashSet<>();
    int n = str.length();

    if (n == 0) {
        permutations.add(prefix);
    } else {
        for (int i = 0; i < n; i++) {
            permutations.addAll(permuteAndStore(prefix + str.charAt(i),
                str.substring(i + 1, n) + str.substring(0, i)));
        }
    }

    return permutations;
}
```

Например, если переданной строкой является `TEST`, то `Set` в результате выдаст следующие данные (все они являются уникальными перестановками):

```
ETST SETT TEST TTSE STTE STET TETS TSTE TSET TTES ESTT ETTS
```

Использование класса `List` вместо `Set` приведет вот к такому результату (обратите внимание на повторы):

```
TEST TETS TSTE TSET TTES TTSE ESTT ESTT ETTS ETST ETST ETTS STTE STET
STET STTE SETT SETT TTES TTSE TEST TETS TSTE TSET
```

Всего 24 перестановки. Число получаемых перестановок легко выяснить, вычислив факториал числа n ($n!$). Для $n = 4$ (длина строки) $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$. В рекурсивной форме это записывается следующим образом: $n! = n \cdot (n - 1)!$.



Поскольку $n!$ очень быстро приводит к высоким числам (например, $10! = 3\,628\,800$), рекомендуется избегать хранения результатов. Для 10-символьной строки, такой как `HELICOPTER`, существует 3 628 800 перестановок!

Попытка имплементировать решение этой задачи в функциональном стиле Java 8 приведет примерно к следующему ниже фрагменту кода:

```
private static void permuteAndPrintStream(String prefix, String str) {
    int n = str.length();

    if (n == 0) {
        System.out.print(prefix + " ");
    } else {
        IntStream.range(0, n)
            .parallel()
            .forEach(i -> permuteAndPrintStream(prefix + str.charAt(i),
                str.substring(i + 1, n) + str.substring(0, i)));
    }
}
```

Вариант решения, который возвращает `Stream<String>`, в качестве бонуса доступен в исходном коде, прилагаемом к этой книге.

11. Проверка, что строка является палиндромом

Для справки — *палиндром* (будь то строка или число) не изменится, если его перевернуть. Это означает, что обработка (чтение) палиндрома может выполняться с обеих сторон, и будет получен один и тот же результат (например, слово `madam` является палиндромом, а слово `madame` — нет).

Простой в имплементации вариант решения состоит в том, чтобы сравнивать буквы заданной строки, используя подход *"встретиться в середине"*. В общих чертах, это решение сравнивает первый символ с последним, второй символ — с предпоследним и так далее до тех пор, пока не будет достигнута середина строки. Эта имплементация опирается на цикл `while`:

```
public static boolean isPalindrome(String str) {
    int left = 0;
    int right = str.length() - 1;

    while (right > left) {
        if (str.charAt(left) != str.charAt(right)) {
            return false;
        }

        left++;
        right--;
    }
    return true;
}
```

Более сжатый подход к решению задачи будет состоять в переписывании предыдущего решения с использованием цикла `for` вместо цикла `while`, как показано ниже:

```
public static boolean isPalindrome(String str) {
    int n = str.length();

    for (int i = 0; i < n / 2; i++) {
        if (str.charAt(i) != str.charAt(n - i - 1)) {
            return false;
        }
    }
    return true;
}
```

Но можно ли свести это решение к одной строке кода? Ответ — да.

В API Java предусмотрен класс `StringBuilder`, который содержит метод `reverse()`. Как следует из его названия, метод `reverse()` возвращает зеркальную строку. В случае палиндрома заданная строка должна быть равна ее зеркальной версии:

```
public static boolean isPalindrome(String str) {
    return str.equals(new StringBuilder(str).reverse().toString());
}
```

В функциональном стиле Java 8 для этого тоже имеется одна-единственная строка кода. Нужно просто определить целочисленный поток `IntStream` в диапазоне от 0 до половины заданной строки и использовать *закрывающую* терминальную операцию `noneMatch()` с предикатом, который сравнивает буквы, следуя подходу *"встретиться в середине"*:

```
public static boolean isPalindrome(String str) {
    return IntStream.range(0, str.length() / 2)
        .noneMatch(p -> str.charAt(p) !=
            str.charAt(str.length() - p - 1));
}
```

Теперь давайте поговорим об удалении повторяющихся символов из заданной строки.

12. Удаление повторяющихся символов

Давайте начнем с варианта решения этой задачи, который опирается на класс `StringBuilder`. В общих чертах, это решение должно перебирать символы заданной строки в цикле и конструировать новую строку, содержащую уникальные символы (просто удалить символы из заданной строки не получится, т. к. в среде Java строка является немутуируемой).

Класс `StringBuilder` выставляет наружу метод с именем `indexOf()`, который возвращает индекс первого появления указанной подстроки (в нашем случае заданного символа) в заданной строке. Таким образом, потенциальным вариантом решения этой задачи будет перебор символов заданной строки в цикле и добавление их по

одному в `StringBuilder` всякий раз, когда метод `indexOf()`, применяемый к текущему символу, возвращает `-1` (это отрицательное значение означает, что `StringBuilder` не содержит текущий символ):

```
public static String removeDuplicates(String str) {
    char[] chArray = str.toCharArray(); // или использовать charAt(i)
    StringBuilder sb = new StringBuilder();

    for (char ch : chArray) {
        if (sb.indexOf(String.valueOf(ch)) == -1) {
            sb.append(ch);
        }
    }

    return sb.toString();
}
```

Следующий далее вариант решения основан на сотрудничестве между коллекцией `HashSet` и классом `StringBuilder`. В общих чертах, хеш-множество `HashSet` обеспечивает устранение повторов, тогда как `StringBuilder` сохраняет полученную строку. Если метод `HashSet.add()` возвращает `true`, мы добавляем символ в `StringBuilder`:

```
public static String removeDuplicates(String str) {
    char[] chArray = str.toCharArray();
    StringBuilder sb = new StringBuilder();
    Set<Character> chHashSet = new HashSet<>();

    for (char c: chArray) {
        if (chHashSet.add(c)) {
            sb.append(c);
        }
    }
    return sb.toString();
}
```

Варианты решения, которые мы представили до этого, используют метод `toCharArray()`, чтобы конвертировать заданную строку в массив `char[]`. В качестве альтернативы оба варианта решения также могут использовать метод `str.charAt(position)`.

Третий вариант решения основан на функциональном стиле Java 8:

```
public static String removeDuplicates(String str) {
    return Arrays.asList(str.split("")).stream()
        .distinct()
        .collect(Collectors.joining());
}
```

Сначала этот вариант решения конвертирует заданную строку в `Stream<String>`, где каждый элемент фактически является одним символом. Далее этот вариант реше-

ния применяет промежуточную операцию `distinct()` с сохранением состояния. Эта операция устраняет из потока повторы, поэтому она возвращает поток без дубликатов. Наконец, этот вариант решения вызывает заключительную операцию `collect()` и опирается на метод `Collectors.joining()`, который просто конкатенирует символы в строку в порядке появления.

13. Удаление заданного символа

Вариант решения этой задачи, который опирается на поддержку JDK, может использовать метод `String.replaceAll()`. Данный метод заменяет каждую подстроку (в нашем случае каждый символ) заданной строки, которая совпадает с заданным регулярным выражением (в нашем случае регулярным выражением является сам символ) с заданной заменой (в нашем случае заменой является пустая строка `""`):

```
public static String removeCharacter(String str, char ch) {
    return str.replaceAll(Pattern.quote(String.valueOf(ch)), "");
}
```

Обратите внимание, что регулярное выражение обернуто в метод `Pattern.quote()`. Это необходимо для экранирования специальных символов, таких как `<`, `(`, `[`, `{`, `\`, `^`, `-`, `=`, `$`, `!`, `|`, `]`, `}`, `,`, `?`, `*`, `+`, `.` и `>`. В общих чертах этот метод возвращает строку literalного шаблона для заданной строки.

Теперь давайте рассмотрим вариант решения, который позволяет избежать регулярных выражений. На этот раз решение задачи опирается на `StringBuilder`. В сущности, этот вариант решения перебирает символы заданной строки в цикле и сравнивает каждый символ с удаляемым символом. Всякий раз, когда текущий символ отличается от удаляемого символа, текущий символ добавляется в `StringBuilder`:

```
public static String removeCharacter(String str, char ch) {
    StringBuilder sb = new StringBuilder();
    char[] chArray = str.toCharArray();

    for (char c : chArray) {
        if (c != ch) {
            sb.append(c);
        }
    }
    return sb.toString();
}
```

Наконец, давайте сосредоточимся на подходе на основе функционального стиля Java 8. Этот подход состоит из четырех шагов:

1. Конвертировать строку в `IntStream` посредством метода `String.chars()`.
2. Отфильтровать `IntStream`, чтобы устранить повторы.
3. Отобразить результирующий `IntStream` в `Stream<String>`.
4. Соединить строки из этого потока и собрать их в единую строку.

Исходный код этого варианта решения можно написать следующим образом:

```
public static String removeCharacter(String str, char ch) {
    return str.chars()
        .filter(c -> c != ch)
        .mapToObj(c -> String.valueOf((char) c))
        .collect(Collectors.joining());
}
```

В качестве альтернативы, если мы хотим удалить юникодовую суррогатную пару, мы можем опереться на методы `codePointAt()` и `codePoints()`, как показано в следующей ниже имплементации:

```
public static String removeCharacter(String str, String ch) {
    int codePoint = ch.codePointAt(0);

    return str.codePoints()
        .filter(c -> c != codePoint)
        .mapToObj(c -> String.valueOf(Character.toChars(c)))
        .collect(Collectors.joining());
}
```



Относительно поддержки сторонними библиотеками, пожалуйста, обратитесь к библиотеке `Apache Commons Lang` и ее методу `StringUtils.remove()`.

Теперь давайте поговорим о том, как найти символ с наибольшим числом появлений.

14. Отыскание символа с наибольшим числом появлений

Довольно простой вариант решения опирается на коллекцию `HashMap` и состоит из трех шагов:

1. Во-первых, перебрать символы заданной строки в цикле и поместить пары "ключ—значение" в коллекцию `HashMap`, где ключ — это текущий символ, а значение — текущее число появлений.
2. Во-вторых, вычислить максимальное значение в коллекции `HashMap` (например, используя `Collections.max()`), представляющий максимальное число появлений.
3. Наконец, получить символ, который имеет максимальное число появлений, обойдя в цикле множество элементов коллекции `HashMap`.

Служебный метод возвращает экземпляр класса `Pair<Character, Integer>`, содержащий символ с наибольшим числом появлений и числом появлений (обратите внимание, что пробелы игнорируются). Если вы не хотите иметь этот дополнительный класс, т. е. `Pair`, то просто обопритесь на `Map.Entry<K, V>`:

```
public static Pair<Character, Integer> maxOccurenceCharacter(String str) {
    Map<Character, Integer> counter = new HashMap<>();
    char[] chStr = str.toCharArray();
```

```

for (int i = 0; i < chStr.length; i++) {
    char currentCh = chStr[i];
    if (!Character.isWhitespace(currentCh)) { // игнорирование пробелов
        Integer noCh = counter.get(currentCh);
        if (noCh == null) {
            counter.put(currentCh, 1);
        } else {
            counter.put(currentCh, ++noCh);
        }
    }
}

int maxOccurrences = Collections.max(counter.values());
char maxCharacter = Character.MIN_VALUE;

for (Entry<Character, Integer> entry: counter.entrySet()) {
    if (entry.getValue() == maxOccurrences) {
        maxCharacter = entry.getKey();
    }
}

return Pair.of(maxCharacter, maxOccurrences);
}

```

Если применение коллекции `HashMap` выглядит громоздким, то еще один вариант решения (который работает немного быстрее) состоит в использовании кодов ASCII. Этот вариант решения начинается с пустого массива из 256 индексов (число 256 — это максимальное число расширенных кодов таблицы ASCII; дополнительную информацию можно найти в *разд. "2. Отыскание первого неповторяющегося символа" ранее в этой главе*). Далее это решение перебирает символы заданной строки в цикле и отслеживает число появлений каждого символа, увеличивая соответствующий индекс в этом массиве:

```

private static final int EXTENDED_ASCII_CODES = 256;
...
public static Pair<Character, Integer> maxOccurrenceCharacter(String str) {
    int maxOccurrences = -1;
    char maxCharacter = Character.MIN_VALUE;
    char[] chStr = str.toCharArray();
    int[] asciiCodes = new int[EXTENDED_ASCII_CODES];

    for (int i = 0; i < chStr.length; i++) {
        char currentCh = chStr[i];
        if (!Character.isWhitespace(currentCh)) { // игнорирование пробела
            int code = (int) currentCh;
            asciiCodes[code]++;
            if (asciiCodes[code] > maxOccurrences) {

```

```
        maxOccurrences = asciiCodes[code];
        maxCharacter = currentCh;
    }
}

return Pair.of(maxCharacter, maxOccurrences);
}
```

Последний вариант решения задачи, который мы обсудим здесь, опирается на функциональный стиль Java 8:

```
public static Pair<Character, Long> maxOccurenceCharacter(String str) {
    return str.chars()
        .filter(c -> Character.isWhitespace(c) == false) // игнорирование пробела
        .mapToObj(c -> (char) c)
        .collect(groupingBy(c -> c, counting()))
        .entrySet()
        .stream()
        .max(comparingByValue())
        .map(p -> Pair.of(p.getKey(), p.getValue()))
        .orElse(Pair.of(Character.MIN_VALUE, -1L));
}
```

Для начала этот вариант решения собирает в коллекции Map отдельные символы в качестве ключей и число их появлений в качестве значений. Далее он использует терминальные операции Java 8 Map.Entry.comparingByValue() и max(), чтобы выяснить элемент отображения с наибольшим значением (наибольшим числом появлений). Поскольку метод max() является терминальной операцией, этот вариант решения возвращает объект Optional<Entry<Character, Long>>, тем самым добавляя лишний шаг, который состоит в отображении этого элемента в Pair<Character, Long>.

15. Сортировка массива строк по длине

Первое, что приходит на ум при сортировке, — это использование компаратора.

В данном случае решение задачи предполагает сравнение длин строк, и поэтому целые числа возвращаются в результате вызова метода String.length() для каждой строки в заданном массиве. И если целые числа отсортированы (по возрастанию или по убыванию), то строки будут отсортированы.

В классе Arrays среды Java уже предусмотрен метод sort(), который берет подлежащий сортировке массив и компаратор. В данном случае эту работу должен сделать Comparator<String>.



До Java 7 код, имплементировавший компаратор, опирался на метод compareTo(). Этот метод очень часто использовался для вычисления разности типа $x_1 - x_2$, но это вычисление может приводить к переполнениям, в результате чего использовать compareTo() становится довольно утомительно. Начиная с Java 7, рекомендуется применять метод Integer.compare() (без риска переполнения).

Ниже приведено решение, которое сортирует заданный массив, опираясь на метод `Arrays.sort()`:

```
public static void sortArrayByLength(String[] strs, Sort direction) {
    if (direction.equals(Sort.ASC)) {
        Arrays.sort(strs, (String s1, String s2)
            -> Integer.compare(s1.length(), s2.length()));
    } else {
        Arrays.sort(strs, (String s1, String s2)
            -> (-1) * Integer.compare(s1.length(), s2.length()));
    }
}
```



Каждая обертка примитивного числового типа имеет свой метод `compare()`.

Начиная с Java 8, интерфейс `Comparator` обогащен значительным числом полезных методов. Одним из таких методов является `comparingInt()`, принимающий функцию. Эта функция извлекает сортировочный ключ типа `int` из обобщенного типа и возвращает значение `Comparator<T>`, которое сравнивает его с этим сортировочным ключом. Еще одним полезным методом является метод `reversed()`, который инвертирует текущее значение компаратора `Comparator`.

На основе этих двух методов мы можем расширить способности метода `Arrays.sort()` следующим образом:

```
public static void sortArrayByLength(String[] strs, Sort direction) {
    if (direction.equals(Sort.ASC)) {
        Arrays.sort(strs, Comparator.comparingInt(String::length));
    } else {
        Arrays.sort(strs,
            Comparator.comparingInt(String::length).reversed());
    }
}
```



Компараторы можно выстраивать в цепочку посредством метода `thenComparing()`.

Представленные здесь варианты решения возвращают `void`, и это означает, что они сортируют заданный массив. Для того чтобы вернуть новый отсортированный массив и не изменять заданный массив, мы можем применить функциональный стиль Java 8, как показано в следующем фрагменте кода:

```
public static String[] sortArrayByLength(String[] strs, Sort direction) {
    if (direction.equals(Sort.ASC)) {
        return Arrays.stream(strs)
            .sorted(Comparator.comparingInt(String::length))
            .toArray(String[]::new);
    }
}
```

```
} else {  
    return Arrays.stream(strs)  
        .sorted(Comparator.comparingInt(String::length).reversed())  
        .toArray(String[]::new);  
}  
}
```

Таким образом, этот фрагмент кода создает поток из заданного массива, сортирует его с помощью промежуточной операции `sorted()` с сохранением состояния и собирает результат в еще одном массиве.

16. Проверка наличия подстроки в строке

Очень простой однострочный вариант решения опирается на метод `String.contains()`.

Этот метод возвращает значение типа `boolean`, указывающее на присутствие или отсутствие заданной подстроки в строке:

```
String text = "hello world!";  
String subtext = "orl";
```

```
// обратите внимание, что будет возвращено true для subtext=""  
boolean contains = text.contains(subtext);
```

В качестве альтернативы этот вариант решения можно имплементировать, опираясь на метод `String.indexOf()` (или `String.lastIndexOf()`) следующим образом:

```
public static boolean contains(String text, String subtext) {  
    return text.indexOf(subtext) != -1; // или lastIndexOf()  
}
```

Еще один вариант решения может быть имплементирован на основе регулярного выражения следующим образом:

```
public static boolean contains(String text, String subtext) {  
    return text.matches("(?i).*" + Pattern.quote(subtext) + ".*");  
}
```

Обратите внимание, что регулярное выражение обернуто в метод `Pattern.quote()`. Это необходимо для экранирования специальных символов, таких как `<([{\^-= $!|})?*.+>` в заданной подстроке.



Относительно поддержки сторонними библиотеками, пожалуйста, обратитесь к библиотеке `Apache Commons Lang` и ее методу `StringUtils.containsIgnoreCase()`.

17. Подсчет числа появлений подстроки в строке

Задача подсчета числа появлений строки в другой строке может иметь по крайней мере две интерпретации:

- ◆ 11 в 111 появляется 1 раз;
- ◆ 11 в 111 появляется 2 раза.

В первом случае (11 в 111 появляется 1 раз) решение задачи может опираться на метод `String.indexOf()`. Одна из разновидностей этого метода позволяет получать индекс первого появления указанной подстроки внутри этой строки, начиная с указанного индекса (или `-1`, если такого появления нет). Основываясь на этом методе, решение задачи может просто обходить заданную строку в цикле и подсчитывать появления заданной подстроки. Обход начинается с позиции 0 и продолжается до тех пор, пока подстрока не будет найдена:

```
public static int countStringInString(String string, String toFind) {
    int position = 0;
    int count = 0;
    int n = toFind.length();

    while ((position = string.indexOf(toFind, position)) != -1) {
        position = position + n;
        count++;
    }
    return count;
}
```

В качестве альтернативы этот вариант решения может использовать метод `String.split()`. В сущности, этот вариант решения разбивает заданную строку, используя заданную подстроку в качестве разделителя. Длина результирующего массива `String[]` должна быть равна числу ожидаемых появлений:

```
public static int countStringInString(String string, String toFind) {
    int result = string.split(Pattern.quote(toFind), -1).length - 1;

    return result < 0 ? 0 : result;
}
```

Во втором случае (11 в 111 появляется 2 раза) решение может опираться на классы `Pattern` и `Matcher` в простой имплементации, как показано ниже:

```
public static int countStringInString(String string, String toFind) {
    Pattern pattern = Pattern.compile(Pattern.quote(toFind));
    Matcher matcher = pattern.matcher(string);

    int position = 0;
    int count = 0;

    while (matcher.find(position)) {
        position = matcher.start() + 1;
        count++;
    }
    return count;
}
```

Прекрасно! Давайте продолжим еще одной задачей со строками.

18. Проверка, являются ли две строки анаграммами

Две строки, имеющие одинаковые символы, но расположенные в другом порядке, являются анаграммами. Иногда в условии задачи уточняется, что анаграммы нечувствительны к регистру и/или пробелы (пустые места) игнорируются.

Таким образом, независимо от применяемого алгоритма, решение данной задачи должно конвертировать заданную строку в нижний регистр и удалять пробелы (пустые места). Кроме того, первый вариант решения, который мы упомянули, сортирует массивы посредством метода `Arrays.sort()` и будет проверять эквивалентность строк посредством метода `Arrays.equals()`.

После сортировки, если строки являются анаграммами, они будут эквивалентны. (На рис. 1.2 показаны два слова, которые являются анаграммами.)



Рис. 1.2

Данный вариант решения (включая его версию в функциональном стиле Java 8) имеется в исходном коде, прилагаемом к этой книге. Главный недостаток этих двух вариантов решения находится в сортировочной части. Следующий ниже вариант решения устраняет этот шаг и опирается на пустой массив (изначально содержащий только 0) из 256 индексов (расширенные коды символов таблицы ASCII — дополнительную информацию можно найти в разд. "2. Отыскание первого неповторяющегося символа" ранее в этой главе).

Алгоритм этого варианта решения является довольно простым:

- ◆ увеличить значение каждого символа из первой строки в этом массиве, соответствующее коду ASCII, на 1;
- ◆ уменьшить значение каждого символа из второй строки в этом массиве, соответствующее коду ASCII, на 1.

Исходный код выглядит следующим образом:

```
private static final int EXTENDED_ASCII_CODES = 256;
...
public static boolean isAnagram(String str1, String str2) {
    int[] chCounts = new int[EXTENDED_ASCII_CODES];
    char[] chStr1 = str1.replaceAll("\\s", "").toLowerCase().toCharArray();
    char[] chStr2 = str2.replaceAll("\\s", "").toLowerCase().toCharArray();
```

```

if (chStr1.length != chStr2.length) {
    return false;
}

for (int i = 0; i < chStr1.length; i++) {
    chCounts[chStr1[i]]++;
    chCounts[chStr2[i]]--;
}

for (int i = 0; i < chCounts.length; i++) {
    if (chCounts[i] != 0) {
        return false;
    }
}

return true;
}

```

Если в конце этого обхода заданные строки являются анаграммами, то этот массив содержит только 0.

19. Объявление многострочных строковых литералов (или текстовых блоков)

На момент написания этой книги в JDK 12 имелось предложение по добавлению многострочных строковых литералов, так называемое предложение *"JEP 326: сырые строковые литералы"*. Но в последнюю минуту оно было отклонено.

Начиная с JDK 13, эта идея была пересмотрена и, в отличие от отклоненных сырых строковых литералов, текстовые блоки окружаются тремя двойными кавычками `"""`, как показано ниже:

```
String text = """Моя школа, Иллинойская Академия
математики и естественных наук, показала мне, что
в этом мире все может быть и что никогда
не слишком рано начинать мыслить масштабно.""";
```



Текстовые блоки бывают очень полезны при написании многострочных SQL-инструкций, использовании разнообразных языков и т. д. Более подробную информацию о них можно найти по адресу <https://openjdk.java.net/jeps/355>.

Тем не менее существует несколько суррогатных решений, которые можно применять в версиях до JDK 13. У этих решений есть общий момент — использование разделителя строк текста:

```
private static final String LS = System.lineSeparator();
```


Начиная с JDK 8, решение задачи может опираться на метод `String.join()` следующим образом:

```
String text = String.join(LS,
    "Моя школа, Иллинойская Академия,",
    "математики и естественных наук, показала мне, что,",
    "в этом мире все может быть и что никогда",
    "не слишком рано начинать мыслить масштабно.");
```

До JDK 8 элегантный вариант решения, возможно, опирался на метод `StringBuilder`. Этот вариант решение имеется в исходном коде, прилагаемом к данной книге.

В то время как приведенные выше варианты решения хорошо подходят для относительно большого числа строк текста, следующие два хороши в том случае, если у нас лишь несколько строк. Первый из них использует оператор `+`:

```
String text = "Моя школа, Иллинойская Академия," + LS +
    "математики и естественных наук, показала мне, что," + LS +
    "в этом мире все может быть и что никогда" + LS +
    "не слишком рано начинать мыслить масштабно.";
```

Второй базируется на методе `String.format()`:

```
String text = String.format("%s" + LS + "%s" + LS + "%s" + LS + "%s",
    "Моя школа, Иллинойская Академия,",
    "математики и естественных наук, показала мне, что,",
    "в этом мире все может быть и что никогда",
    "не слишком рано начинать мыслить масштабно.");
```



Как обработать каждую строку многострочного строкового литерала? Дело в том, что быстрый подход требует JDK 11, который идет в комплекте с методом `String.lines()`. Этот метод разбивает заданную строку текста посредством разделителя строк текста (который поддерживает `\n`, `\r` и `\r\n`) и трансформирует ее в `Stream<String>`. В качестве альтернативы также может использоваться метод `String.split()` (он доступен, начиная с JDK 1.4). Если число строк текста становится значительным, то рекомендуется поместить их в файл и читать/обрабатывать их по одной (например, посредством метода `getResourceAsStream()`). Другие подходы опираются на `StringWriter` или `BufferedWriter.newLine()`.

Относительно поддержки сторонними библиотеками, пожалуйста, обратитесь к библиотеке `Apache Commons Lang` и ее методу `StringUtils.join()`, библиотеке `Guava` и ее классу `Joiner`, а также к специализированной аннотации `@Multiline`.

20. Конкатенирование одной и той же строки *n* раз

До JDK 11 решение этой задачи можно было быстро предоставить посредством `StringBuilder` следующим образом:

```
public static String concatRepeat(String str, int n) {
    StringBuilder sb = new StringBuilder(str.length() * n);
```

```

for (int i = 1; i <= n; i++) {
    sb.append(str);
}

return sb.toString();
}

```

Начиная с JDK 11, решение опирается на метод `String.repeat(int count)`. Этот метод возвращает строку, получающуюся в результате конкатенирования строки `count` раз. За кулисами этот метод использует метод `System.arraycopy()`, что делает его очень быстрым:

```
String result = "hello".repeat(5);
```

Ниже перечислены другие варианты решения, которые могут хорошо вписываться в различные сценарии:

◆ вариант решения на основе метода `String.join()`:

```
String result = String.join("", Collections.nCopies(5, TEXT));
```

◆ вариант решения на основе метода `Stream.generate()`:

```
String result = Stream.generate(() -> TEXT)
    .limit(5)
    .collect(joining());
```

◆ вариант решения на основе метода `String.format()`:

```
String result = String.format("%0" + 5 + "d", 0)
    .replace("0", TEXT);
```

◆ вариант решения на основе массива `char[]`:

```
String result = new String(new char[5]).replace("\0", TEXT);
```



Относительно поддержки сторонними библиотеками, пожалуйста, обратитесь к библиотеке `Apache Commons Lang` и ее методу `StringUtils.repeat()` и к библиотеке `Guava` и ее методу `Strings.repeat()`.

Для проверки, является ли строка последовательностью одной и той же подстроки, следует использовать такой метод:

```

public static boolean hasOnlySubstrings(String str) {
    StringBuilder sb = new StringBuilder();

    for (int i = 0; i < str.length() / 2; i++) {
        sb.append(str.charAt(i));
        String resultStr = str.replaceAll(sb.toString(), "");
        if (resultStr.length() == 0) {
            return true;
        }
    }

    return false;
}

```

Этот решение обходит в цикле половину заданной строки и поступательно заменяет ее на "", при этом подстрока строится путем посимвольного добавления исходной строки в `StringBuilder`. Если эти замены в результате дают пустую строку, то значит заданная строка является последовательностью одной и той же подстроки.

21. Удаление начальных и замыкающих пробелов

Самое быстрое решение этой задачи, вероятно, опирается на метод `String.trim()`. Этот метод способен удалить все начальные и замыкающие пробелы, т. е. любой символ, кодовая точка которого меньше или равна `U+0020` или 32 (символу пробела):

```
String text = "\n\n\n hello \t\n\r";
String trimmed = text.trim();
```

Приведенный выше фрагмент кода будет работать, как и ожидалось. Обрезанной строкой будет слово `hello`. Это работает только потому, что все используемые пробелы меньше или равны `U+0020` или 32 (символу пробела). В качестве пробелов определено 25 символов (https://en.wikipedia.org/wiki/Whitespace_character#Unicode), и метод `trim()` охватывает только часть из них (одним словом, `trim()` не знает Юникода). Рассмотрим следующую строку:

```
char space = '\u2002';
String text = space + "\n\n\n hello \t\n\r" + space;
```

`\u2002` — это еще один тип пробела, который метод `trim()` не распознает (`\u2002` выше `\u0020`). Это означает, что в таких случаях метод `trim()` не будет работать надлежащим образом. Начиная с `JDK 11`, у проблемы есть решение в виде метода `strip()`, который расширяет возможности метода `trim()` в области Юникода:

```
String stripped = text.strip();
```

На этот раз все начальные и замыкающие пробелы будут удалены.



Более того, `JDK 11` поставляется в комплекте с двумя вариантами метода `strip()` для удаления только начальных (`stripLeading()`) или только замыкающих (`stripTrailing()`) пробелов. Метод `trim()` таких разновидностей не имеет.

22. Поиск наибольшего общего префикса

Рассмотрим следующий массив строк:

```
String[] texts = {"abc", "abcd", "abcde", "ab", "abcd", "abcdef"};
```

Теперь давайте разместим эти строки одну под другой:

```
abc
abcd
abcde

ab
abcd
abcdef
```

Простое сравнение этих строк показывает, что `ab` является наибольшим общим префиксом. Теперь займемся решением этой задачи. Вариант решения, который мы здесь представили, основан на простом сравнении. В нем из массива берется первая строка, и каждый ее символ сравнивается с символами в остальных строках. Алгоритм останавливается, если происходит одно из следующих событий:

- ◆ длина первой строки больше, чем длина любой другой строки;
- ◆ текущий символ первой строки не совпадает с текущим символом любой другой строки.

Если алгоритм принудительно останавливается из-за одного из приведенных выше сценариев, то наибольшим общим префиксом является подстрока длиной от 0 до индекса текущего символа из первой строки. В противном случае наибольшим общим префиксом является первая строка из массива. Исходный код этого решения выглядит следующим образом:

```
public static String longestCommonPrefix(String[] strs) {
    if (strs.length == 1) {
        return strs[0];
    }

    int firstLen = strs[0].length();

    for (int prefixLen = 0; prefixLen < firstLen; prefixLen++) {
        char ch = strs[0].charAt(prefixLen);
        for (int i = 1; i < strs.length; i++) {
            if (prefixLen >= strs[i].length()
                || strs[i].charAt(prefixLen) != ch) {
                return strs[i].substring(0, prefixLen);
            }
        }
    }

    return strs[0];
}
```

Другие варианты решения этой задачи используют хорошо известные алгоритмы, такие как *двоичный поиск* или *префиксное дерево* (так называемое дерево *trie*). В исходном коде, который сопровождает эту книгу, также есть вариант решения, основанный на двоичном поиске.

23. Применение отступа

Начиная с JDK 12, мы можем выделять текст отступами посредством метода `String.indent(int n)`.

Допустим, что у нас есть следующие значения типа `String`:

```
String days = "Sunday\n"  
+ "Monday\n"  
+ "Tuesday\n"  
+ "Wednesday\n"  
+ "Thursday\n"  
+ "Friday\n"  
+ "Saturday";
```

Напечатать эти значения с отступом в 10 пробелов можно следующим образом

```
System.out.print(days.indent(10));
```

Результат будет таким, как показано на рис. 1.3.



```
    Sunday  
    Monday  
    Tuesday  
    Wednesday  
    Thursday  
    Friday  
    Saturday
```

Рис. 1.3

Теперь попробуем выделить их каскадным отступом:

```
List<String> days = Arrays.asList("Sunday", "Monday", "Tuesday",  
    "Wednesday", "Thursday", "Friday", "Saturday");
```

```
for (int i = 0; i < days.size(); i++) {  
    System.out.print(days.get(i).indent(i));  
}
```

Результат приведен на рис. 1.4.



```
    Sunday  
    Monday  
    Tuesday  
    Wednesday  
    Thursday  
    Friday  
    Saturday
```

Рис. 1.4

Теперь выделим отступом в зависимости от длины значения:

```
days.stream()  
    .forEachOrdered(d -> System.out.print(d.indent(d.length())));
```

Результат показан на рис. 1.5.



```

    Sunday
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday

```

Рис. 1.5

Как насчет того, чтобы выделить отступами фрагмент HTML-кода? Давайте посмотрим:

```

String html = "<html>";
String body = "<body>";
String h2 = "<h2>";
String text = "Hello world!";
String closeH2 = "</h2>";
String closeBody = "</body>";
String closeHtml = "</html>";

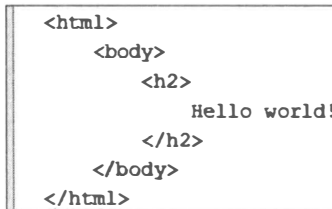
```

```

System.out.println(html.indent(0) + body.indent(4) + h2.indent(8)
    + text.indent(12) + closeH2.indent(8) + closeBody.indent(4)
    + closeHtml.indent(0));

```

Результат показан на рис. 1.6.



```

    <html>
      <body>
        <h2>
          Hello world!
        </h2>
      </body>
    </html>

```

Рис. 1.6

24. Трансформирование строк

Допустим, что у нас есть строка и мы хотим трансформировать ее в еще одну строку (например, перевести ее в верхний регистр). Мы можем сделать это, применив такую функцию, как `Function<? super String, ? extends R>`.

В JDK 8 мы можем выполнить это с помощью метода `map()`, как показано в следующих двух простых примерах:

```

// hello world
String resultMap = Stream.of("hello")

```

```
.map(s -> s + " world")`  
.findFirst()  
.get();  
  
// GOOOOOOOOOOOOOOOOOO! GOOOOOOOOOOOOOOOOOO!  
String resultMap = Stream.of("goooo! ")  
    .map(String::toUpperCase)  
    .map(s -> s.repeat(2))  
    .map(s -> s.replaceAll("O", "OOOO"))  
    .findFirst()  
    .get();
```

Начиная с JDK 12, мы можем опираться на новый метод с именем `transform(Function<? super String, ? extends R> f)`. Давайте перепишем предыдущие фрагменты кода с использованием метода `transform()`:

```
// hello world  
String result = "hello".transform(s -> s + " world");  
  
// GOOOOOOOOOOOOOOOOOO! GOOOOOOOOOOOOOOOOOO!  
String result = "goooo! ".transform(String::toUpperCase)  
    .transform(s -> s.repeat(2))  
    .transform(s -> s.replaceAll("O", "OOOO"));
```

В отличие от более общего метода `map()` метод `transform()` предназначен для применения функции к строке и возвращает результирующую строку.

25. Вычисление минимума и максимума двух чисел

До JDK 8 возможным решением было бы использовать методы `Math.min()` и `Math.max()` следующим образом:

```
int i1 = -45;  
int i2 = -15;  
int min = Math.min(i1, i2);  
int max = Math.max(i1, i2);
```

Класс `Math` предусматривает методы `min()` и `max()` для каждого примитивного числового типа (`int`, `long`, `float` и `double`).

Начиная с JDK 8, каждый оберточный класс примитивных числовых типов (`Integer`, `Long`, `Float` и `Double`) идет в комплекте с выделенными методами `min()` и `max()`, и позади этих методов стоят вызовы соответствующих им методов из класса `Math`. Взгляните на очередной пример (он чуть-чуть выразительнее):

```
double d1 = 0.023844D;  
double d2 = 0.35468856D;  
double min = Double.min(d1, d2);  
double max = Double.max(d1, d2);
```

В контексте функционального стиля потенциальное решение будет опираться на функциональный интерфейс `BinaryOperator`, который идет в комплекте с двумя методами — `minBy()` и `maxBy()`:

```
float f1 = 33.34F;
final float f2 = 33.213F;
float min = BinaryOperator.minBy(Float::compare).apply(f1, f2);
float max = BinaryOperator.maxBy(Float::compare).apply(f1, f2);
```

Оба метода способны возвращать минимум (и соответственно, максимум) двух элементов согласно указанному компаратору.

26. Сложение двух крупных чисел типа *int/long* и переполнение операции

Давайте займемся решением этой задачи, начав с оператора `+`, как в следующем ниже примере:

```
int x = 2;
int y = 7;
int z = x + y;    // 9
```

Этот подход очень прост и прекрасно работает для большинства вычислений с участием типов `int`, `long`, `float` и `double`.

Теперь применим этот оператор к следующим двум большим числам (сумма числа 2 147 483 647 с самим собой):

```
int x = Integer.MAX_VALUE;
int y = Integer.MAX_VALUE;
int z = x + y;    // -2
```

На этот раз переменная `z` будет равна `-2`, что не является ожидаемым результатом, т. е. 4 294 967 294. Изменение типа только переменной `z` с `int` на `long` не поможет. Однако изменение типов переменных `x` и `y` с `int` на `long` даст *ожидаемый* результат:

```
long x = Integer.MAX_VALUE;
long y = Integer.MAX_VALUE;
long z = x + y;    // 4294967294
```

Но проблема появится снова, если вместо `Integer.MAX_VALUE` будет `Long.MAX_VALUE`:

```
long x = Long.MAX_VALUE;
long y = Long.MAX_VALUE;
long z = x + y;    // -2
```

Начиная с JDK 8, оператор `+` был обернут оберткой каждого примитивного числового типа более выразительным образом. И поэтому классы `Integer`, `Long`, `Float` и `Double` имеют метод `sum()`:

```
long z = Long.sum();    // -2
```

За кулисами методы `sum()` тоже используют оператор `+`, поэтому они просто выдают тот же самый результат.

Вместе с тем, также начиная с JDK 8, класс `Math` был дополнен двумя методами `addExact()`. Один метод `addExact()` предназначен для суммирования двух переменных типа `int`, другой — для суммирования двух переменных типа `long`. Оба метода очень полезны, если результат подвержен переполнению типов `int` или `long`, как показано в предыдущем случае. В таких случаях вместо того, чтобы возвращать вводящий в заблуждение результат, методы выбрасывают исключение `ArithmeticException`:

```
int z = Math.addExact(x, y); // выбрасывает исключение ArithmeticException
```

Данный фрагмент кода выбросит исключение, такое как `java.lang.ArithmeticException: integer overflow` (целочисленное переполнение). Оно полезно тем, что позволяет избегать внесения вводящих в заблуждение результатов в дальнейшие вычисления (например, ранее результат `-2` мог бы молча войти в дальнейшие вычисления).

В контексте функционального стиля потенциальное решение этой задачи будет опираться на функциональный интерфейс `BinaryOperator` следующим образом (надо просто определить операцию двух операндов одного типа):

```
BinaryOperator<Integer> operator = Math::addExact;
int z = operator.apply(x, y);
```

Помимо методов `addExact()`, класс `Math` имеет методы `multiplyExact()`, `subtractExact()` и `negateExact()`. Более того, можно управлять хорошо известными инкрементными (`i++`) и декрементными (`i--`) выражениями на предмет переполнения их областей значений посредством методов `incrementExact()` и `decrementExact()` (например, `Math.incrementExact(i)`). Обратите внимание, что эти методы доступны только для типов `int` и `long`.



Во время работы с крупным числом также обратите внимание на класс `BigInteger` (немутируемые целые числа произвольной точности) и класс `BigDecimal` (немутируемые знаковые десятичные числа произвольной точности).

27. Строка как беззнаковое число с основанием системы счисления

Поддержка беззнаковой арифметики была добавлена в Java, начиная с версии 8. Это добавление наибольшее влияние оказало на классы `Byte`, `Short`, `Integer` и `Long`.

В Java строки, представляющие положительные числа, могут быть разобраны как типы `unsigned int` и `long` посредством методов JDK 8 `parseUnsignedInt()` и `parseUnsignedLong()`. Например, рассмотрим следующее целое число в качестве экземпляра типа `String`:

```
String nri = "255500";
```

Вариант решения с его разбором в беззнаковое значение типа `int` с основанием 36 (максимально допустимым основанием) выглядит следующим образом:

```
int result = Integer.parseUnsignedInt(nri, Character.MAX_RADIX);
```

Первый аргумент — это число, а второй — основание. Основание должно находиться в диапазоне [2; 36] или [`Character.MIN_RADIX`; `Character.MAX_RADIX`].

Применить основание 10 можно легко следующим образом (этот метод применяет основание 10 по умолчанию):

```
int result = Integer.parseUnsignedInt(nri);
```

Начиная с JDK 9, метод `parseUnsignedInt()` имеет новую разновидность. Помимо строкового значения и основания, этот метод принимает интервал вида [`beginIndex`, `endIndex`]. На этот раз разбор выполняется в заданном интервале. Например, интервал [1; 3] можно указать следующим образом:

```
int result = Integer.parseUnsignedInt(nri, 1, 4, Character.MAX_RADIX);
```

Метод `parseUnsignedInt()` может разбирать строки, которые представляют числа, превышающие `Integer.MAX_VALUE` (попытка сделать это посредством `Integer.parseInt()` выбросит исключение `java.lang.NumberFormatException`):

```
// Integer.MAX_VALUE + 1 = 2147483647 + 1 = 2147483648
```

```
int maxPlus1 = Integer.parseUnsignedInt("2147483648");
```



Такой же набор методов существует для чисел типа `long` в классе `Long` (например, `parseUnsignedLong()`).

28. Конвертирование в число посредством беззнаковой конверсии

Данная задача требует, чтобы мы конвертировали некоторое значение типа `int` со знаком в тип `long` посредством беззнаковой конверсии. Итак, рассмотрим значение `Integer.MIN_VALUE` со знаком, т. е. -2 147 483 648.

В JDK 8, с применением метода `Integer.toUnsignedLong()` конверсия будет выглядеть следующим образом (результат будет 2 147 483 648):

```
long result = Integer.toUnsignedLong(Integer.MIN_VALUE);
```

Вот еще один пример, который конвертирует значения `Short.MIN_VALUE` и `Short.MAX_VALUE` со знаком в массив беззнаковых целых чисел:

```
int result1 = Short.toUnsignedInt(Short.MIN_VALUE);
```

```
int result2 = Short.toUnsignedInt(Short.MAX_VALUE);
```

Другими методами из той же категории являются методы `Integer.toUnsignedString()`, `Long.toUnsignedString()`, `Byte.toUnsignedInt()`, `Byte.toUnsignedLong()`, `Short.toUnsignedInt()` и `Short.toUnsignedLong()`.

29. Сравнение двух беззнаковых чисел

Рассмотрим два знаковых целых числа, `Integer.MIN_VALUE` ($-2\,147\,483\,648$) и `Integer.MAX_VALUE` ($2\,147\,483\,647$). Если сравнить эти целые числа (знаковые значения), в результате мы получим, что $-2\,147\,483\,648$ будет меньше $2\,147\,483\,647$:

```
// значение resultSigned, равное -1, говорит о том, что
// MIN_VALUE меньше MAX_VALUE
int resultSigned = Integer.compare(Integer.MIN_VALUE, Integer.MAX_VALUE);
```

В JDK 8 эти два целых числа можно сравнить как беззнаковые значения посредством метода `Integer.compareUnsigned()` (он является эквивалентом методу `Integer.compare()` для беззнаковых значений). В общих чертах этот метод игнорирует понятие *знакового бита*, и самый левый бит считается самым значащим битом. Под зонтиком беззнаковых значений указанный метод возвращает 0, если сравниваемые числа равны, возвращает значение меньше 0, если первое беззнаковое значение меньше второго, и возвращает значение больше 0, если первое беззнаковое значение больше второго.

Следующее ниже сравнение возвращает 1, указывая на то, что беззнаковое значение `Integer.MIN_VALUE` больше беззнакового значения `Integer.MAX_VALUE`:

```
// значение resultSigned равное 1 говорит о том, что
// MIN_VALUE больше MAX_VALUE
int resultUnsigned
    = Integer.compareUnsigned(Integer.MIN_VALUE, Integer.MAX_VALUE);
```



Метод `compareUnsigned()` доступен в классах `Integer` и `Long`, начиная с JDK 8, и в классах `Byte` и `Short`, начиная с JDK 9.

30. Вычисление частного и остатка от деления беззнаковых значений

Вычисление беззнакового частного и остатка, полученного в результате деления двух беззнаковых значений, поддерживается API (интерфейсом прикладного программирования) беззнаковой арифметики JDK 8 посредством методов `divideUnsigned()` и `remainderUnsigned()`.

Давайте рассмотрим знаковые числа `Integer.MIN_VALUE` и `Integer.MAX_VALUE` и выполним деление без остатка и с остатком. Здесь нет ничего нового:

```
// знаковое частное
// -1
int divisionSignedMinMax = Integer.MIN_VALUE / Integer.MAX_VALUE;

// 0
int divisionSignedMaxMin = Integer.MAX_VALUE / Integer.MIN_VALUE;
```

```
// знаковый остаток
// -1
int moduloSignedMinMax = Integer.MIN_VALUE % Integer.MAX_VALUE;

// 2147483647
int moduloSignedMaxMin = Integer.MAX_VALUE % Integer.MIN_VALUE;
```

Теперь рассмотрим Integer.MIN_VALUE и Integer.MAX_VALUE как беззнаковые значения и применим divideUnsigned() и remainderUnsigned():

```
// беззнаковое частное
int divisionUnsignedMinMax = Integer.divideUnsigned(
    Integer.MIN_VALUE, Integer.MAX_VALUE); // 1
int divisionUnsignedMaxMin = Integer.divideUnsigned(
    Integer.MAX_VALUE, Integer.MIN_VALUE); // 0

// беззнаковый остаток
int moduloUnsignedMinMax = Integer.remainderUnsigned(
    Integer.MIN_VALUE, Integer.MAX_VALUE); // 1
int moduloUnsignedMaxMin = Integer.remainderUnsigned(
    Integer.MAX_VALUE, Integer.MIN_VALUE); // 2147483647
```

Обратите внимание на их сходство с операцией сравнения. Обе операции, т. е. беззнаковое деление без остатка и беззнаковое деление с остатком, интерпретируют все биты как биты значения и игнорируют бит знака.



Методы divideUnsigned() и remainderUnsigned() присутствуют соответственно в классах Integer и Long.

31. Значение типа *double/float* является конечным значением с плавающей точкой

Эта задача возникает из-за того, что некоторые методы и операции с плавающей точкой иногда возвращают Infinity или NaN вместо выбрасывания исключения.

Решение задачи проверки, является ли заданное число типа float/double конечным значением с плавающей запятой, опирается на следующие ниже условия — абсолютное значение заданного значения типа float/double не должно превышать наибольшее положительное конечное значение типа float/double:

```
// для числа одинарной точности
Math.abs(f) <= Float.MAX_VALUE;

// для числа двойной точности
Math.abs(d) <= Double.MAX_VALUE
```

Начиная с Java 8, приведенные выше условия были выставлены наружу через два выделенных флаговых метода — `Float.isFinite()` и `Double.isFinite()`. Поэтому следующие примеры являются допустимыми тестовыми случаями для конечных значений с плавающей запятой:

```
Float f1 = 4.5f;
boolean f1f = Float.isFinite(f1); // f1 = 4.5, является конечным

Float f2 = f1 / 0;
boolean f2f = Float.isFinite(f2); // f2 = Infinity, не является конечным

Float f3 = 0f / 0f;
boolean f3f = Float.isFinite(f3); // f3 = NaN, не является конечным

Double d1 = 0.000333411333d;
boolean d1f = Double.isFinite(d1); // d1 = 3.33411333E-4, является конечным

Double d2 = d1 / 0;
boolean d2f = Double.isFinite(d2); // d2 = Infinity, не является конечным

Double d3 = Double.POSITIVE_INFINITY * 0;
boolean d3f = Double.isFinite(d3); // d3 = NaN, не является конечным
```

Эти методы удобны в приведенных далее условных конструкциях:

```
if (Float.isFinite(d1)) {
    // выполнить вычисление с конечным значением d1 с плавающей точкой
} else {
    // d1 не может войти в дальнейшие вычисления
}
```

32. Применение логического И/ИЛИ/исключающего ИЛИ к двум булевым выражениям

Таблица истинности элементарных логических операций (**И**, **ИЛИ** и **исключающее ИЛИ**) приведена на рис. 1.7.

X	Y	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Рис. 1.7

В языке Java логический оператор **И** представлен как `&&`, логический оператор **ИЛИ** представлен как `||`, и логический оператор "исключающее **ИЛИ**" представлен как `^`.

Начиная с JDK 8, эти операторы применяются к двум булевым значениям и обертываются в три статических (static) метода — `Boolean.logicalAnd()`, `Boolean.logicalOr()` и `Boolean.logicalXor()`:

```
int s = 10;
int m = 21;

// если (s > m && m < 50) { } иначе { }
if (Boolean.logicalAnd(s > m, m < 50)) {} else {}

// если (s > m || m < 50) { } иначе { }
if (Boolean.logicalOr(s > m, m < 50)) {} else {}

// если (s > m ^ m < 50) { } иначе { }
if (Boolean.logicalXor(s > m, m < 50)) {} else {}
```

Комбинация этих методов также возможна:

```
if (Boolean.logicalAnd(
    Boolean.logicalOr(s > m, m < 50),
    Boolean.logicalOr(s <= m, m > 50))) {} else {}
```

33. Конвертирование значения типа *BigInteger* в примитивный тип

Класс `BigInteger` является очень удобным инструментом для представления немутуируемых целых чисел произвольной точности.

Этот класс также содержит методы (происходящие из `java.lang.Number`), которые полезны для конвертирования `BigInteger` в примитивный тип, такой как `byte`, `long` или `double`. Однако эти методы могут выдавать неожиданные результаты и вносить путаницу. Например, допустим, что у нас есть `BigInteger`, который обертывает `Long.MAX_VALUE`:

```
BigInteger nr = BigInteger.valueOf(Long.MAX_VALUE);
```

Давайте конвертируем этот класс `BigInteger` в число примитивного типа `long` посредством метода `BigInteger.longValue()`:

```
long nrLong = nr.longValue();
```

До сих пор все работало так, как ожидалось, поскольку `Long.MAX_VALUE` равно 9 223 372 036 854 775 807, и переменная `nrLong` примитивного типа `long` имеет именно это значение.

Теперь попробуем конвертировать класс `BigInteger` в значение примитивного типа `int` посредством метода `BigInteger.intValue()`:

```
int nrInt = nr.intValue();
```

На этот раз переменная `nrInt` примитивного типа `long` будет иметь значение `-1` (тот же результат дадут методы `shortValue()` и `byteValue()`). В соответствии с документацией, если значение `BigInteger` является слишком большим для указанного примитивного типа, то возвращаются только младшие *n* бит (*n* зависит от указанного примитивного типа). Но если исходный код не осведомлен об этом утверждении, то в дальнейшие вычисления он будет вводить значения `-1`, что приведет к путанице.

Однако, начиная с JDK 8, был добавлен новый набор методов. Эти методы предназначены для выявления информации, которая теряется при конвертации из `BigInteger` в указанный примитивный тип. Если обнаруживается фрагмент потерянной информации, то будет выбрасываться исключение `ArithmeticException`. Благодаря этому код сигнализирует о том, что конверсия столкнулась с некоторыми проблемами, и предотвращает эту неприятную ситуацию.

Этими методами являются `longValueExact()`, `intValueExact()`, `shortValueExact()` и `byteValueExact()`:

```
long nrExactLong = nr.longValueExact(); // работает как ожидается
int nrExactInt = nr.intValueExact();    // выбрасывает исключение
                                     // ArithmeticException
```

Обратите внимание, что функция `intValueExact()` не возвращает значение `-1` как `intValue()`. На этот раз сигнал о потере информации, вызванной попыткой конвертировать наибольшее значение типа `long` в тип `int`, был подан через исключение `ArithmeticException`.

34. Конвертирование значение типа *long* в значение типа *int*

Задача конвертирования значения типа `long` в значение типа `int` выглядит легкой. Например, потенциальный вариант решения может основываться на приведении типа следующим образом:

```
long nr = Integer.MAX_VALUE;
int intNrCast = (int) nr;
```

В качестве альтернативы решение задачи может опираться на метод `Long.intValue()` следующим образом:

```
int intNrValue = Long.valueOf(nrLong).intValue();
```

Оба варианта работают без проблем. Теперь предположим, что мы имеем следующее значение типа `long`:

```
long nrMaxLong = Long.MAX_VALUE;
```

На этот раз оба варианта вернут `-1`. Для того чтобы избежать таких результатов, рекомендуется опираться на JDK 8, т. е. на метод `Math.toIntExact()`. Этот метод получает аргумент типа `long` и пытается конвертировать его в `int`. Если получаемое

значение переполняет тип `int`, то этот метод будет выбрасывать исключение `ArithmeticException`:

```
// выбрасывает исключение ArithmeticException
int intNrMaxExact = Math.toIntExact(nrMaxLong);
```

За кулисами метод `toIntExact()` опирается на условие `((int)value != value)`.

35. Вычисление целой части деления и целой части остатка от деления

Допустим, у нас есть следующая инструкция с делением:

```
double z = (double)222/14;
```

Эта инструкция инициализирует переменную `z` результатом деления, т. е. 15.85, но наша задача запрашивает целую часть этого деления, т. е. 15 (это самое большое целое значение, которое меньше или равно алгебраическому частному). Решение с получением этого желаемого результата сводится к применению метода `Math.floor(15.85)`, т. е. 15.

Однако 222 и 14 являются целыми числами, и поэтому приведенное выше деление записывается следующим образом:

```
int z = 222/14;
```

На этот раз переменная `z` будет равна 15, что в точности соответствует ожидаемому результату (оператор `/` возвращает ближайшее к нулю целое число). Нет необходимости применять метод `Math.floor(z)`. Более того, если делитель равен 0, то `222/0` выбросит исключение `ArithmeticException`.

В этом месте можно заключить, что целая часть деления двух целых чисел, имеющих один и тот же знак (оба положительные или отрицательные), может быть получена посредством оператора `/`.

ОК, пока все хорошо, но давайте допустим, что у нас есть следующие два целых числа (противоположных знаков; отрицательное делимое и положительный делитель, и наоборот):

```
double z = (double) -222/14;
```

На этот раз переменная `z` будет равна -15.85. После применения метода `Math.floor(z)` результат будет равен -16, что правильно (это самое большое целое значение, которое меньше или равно алгебраическому частному).

Давайте снова вернемся к той же задаче с типом `int`:

```
int z = -222/14;
```

На этот раз `z` будет равно -15. Это неверно, и в данном случае метод `Math.floor(z)` нам не поможет, т. к. `Math.floor(-15)` равно -15. Поэтому данную проблему следует рассмотреть особо.

Начиная с JDK 8, все эти случаи были охвачены и выставлены наружу посредством метода `Math.floorDiv()`. Этот метод в качестве аргументов берет два целых числа, представляющих делимое и делитель, и возвращает наибольшее (ближайшее к по-

ложительной бесконечности) значение типа `int`, которое меньше или равно алгебраическому частному:

```
int x = -222;
int y = 14;

// x является делимым, y является делителем
int z = Math.floorDiv(x, y);    // -16
```

Метод `Math.floorDiv()` идет в трех разновидностях: `floorDiv(int x, int y)`, `floorDiv(long x, int y)` и `floorDiv(long x, long y)`.



После метода `Math.floorDiv()` среда разработки JDK 8 предоставила метод `Math.floorMod()`, который возвращает целую часть остатка от деления заданных аргументов. Она вычисляется как результат выражения $x - (\text{floorDiv}(x, y) * y)$, и поэтому этот метод вернет тот же результат, что и оператор `%` для аргументов с одинаковым знаком и другой результат для аргументов, которые не имеют одинакового знака.

Округлить вверх результат деления двух натуральных чисел (a/b) можно следующим образом:

```
long result = (a + b - 1) / b;
```

Ниже приведен один пример такого округления (у нас $4/3 = 1.33$, и мы хотим 2):

```
long result = (4 + 3 - 1) / 3;    // 2
```

Ниже приведен еще один пример округления (у нас $17/7 = 2.42$, и мы хотим 3):

```
long result = (17 + 7 - 1) / 7;    // 3
```

Если целые числа не являются положительными, то мы можем опереться на метод `Math.ceil()`:

```
long result = (long) Math.ceil((double) a/b);
```

36. Следующее значение с плавающей точкой

Имея целочисленное значение, такое как 10, нам легко получить следующее целочисленное значение, такое как $10+1$ (в направлении положительной бесконечности) или $10-1$ (в направлении отрицательной бесконечности). Достичь того же самого для значения типа `float` или `double` будет не так просто, как для целых чисел.

Начиная с JDK 6, класс `Math` дополнен методом `nextAfter()`. Этот метод берет два аргумента — начальное число (типа `float` или `double`) и направление (`Float/Double.NEGATIVE/POSITIVE_INFINITY`) — и возвращает следующее значение с плавающей точкой. Здесь используется разновидность данного метода, которая возвращает следующую плавающую точку, смежную с 0,1 в направлении отрицательной бесконечности:

```
float f = 0.1f;

// 0.099999994
float nextf = Math.nextAfter(f, Float.NEGATIVE_INFINITY);
```

Начиная с JDK 8, класс `Math` дополнен двумя методами, которые действуют как укороченные формы метода `nextAfter()` и являются более быстрыми. Этими методами являются `nextDown()` и `nextUp()`:

```
float f = 0.1f;

float nextdownf = Math.nextDown(f);    // 0.099999994
float nextupf = Math.nextUp(f);        // 0.10000001

double d = 0.1d;

double nextdownnd = Math.nextDown(d);  // 0.09999999999999999
double nextupnd = Math.nextUp(d);      // 0.10000000000000002
```

Поэтому метод `nextAfter()` в направлении отрицательной бесконечности доступен посредством методов `Math.nextDown()` и `nextAfter()`, тогда как в направлении положительной бесконечности он доступен посредством метода `Math.nextUp()`.

37. Умножение двух крупных значений типа *int/long* и переполнение операции

Давайте займемся решением этой задачи, начав с оператора `*`, как показано в следующем ниже примере:

```
int x = 10;
int y = 5;
int z = x * y;    // 50
```

Этот подход очень прост и прекрасно работает для большинства вычислений с участием значения типа `int`, `long`, `float` и `double`.

Теперь применим этот оператор к следующим двум крупным числам (умножим число 2 147 483 647 на себя):

```
int x = Integer.MAX_VALUE;
int y = Integer.MAX_VALUE;
int z = x * y;    // 1
```

На этот раз переменная `z` будет равна 1, что не является ожидаемым результатом, т. е. числом 4 611 686 014 132 420 609. Изменение типа только переменной `z` с `int` на `long` не поможет. А вот изменение типов переменных `x` и `y` с `int` на `long` даст ожидаемый результат:

```
long x = Integer.MAX_VALUE;
long y = Integer.MAX_VALUE;
long z = x * y;    // 4611686014132420609
```

Но проблема возникнет снова, если мы имеем `Long.MAX_VALUE` вместо `Integer.MAX_VALUE`:

```
long x = Long.MAX_VALUE;
long y = Long.MAX_VALUE;
long z = x * y;    // 1
```

Таким образом, вычисления, которые переполняют область значений и опираются на оператор `*`, в итоге приведут к ошибочным результатам.

Вместо того чтобы использовать эти результаты в дальнейших вычислениях, лучше быть проинформированным вовремя, когда произошло переполнение операции. JDK 8 идет в комплекте с методом `Math.multiplyExact()`. Этот метод пытается перемножить два целых числа. Если в результате тип `int` переполняется, то он будет просто выбрасывать исключение `ArithmeticException`:

```
int x = Integer.MAX_VALUE;
int y = Integer.MAX_VALUE;
int z = Math.multiplyExact(x, y); // выбрасывает исключение ArithmeticException
```



В JDK 8 метод `Math.multiplyExact(int x, int y)` возвращает тип `int` и метод `Math.multiplyExact(long x, long y)` возвращает тип `long`. В JDK 9 был также добавлен метод `Math.multiplyExact(long, int y)`, возвращающий тип `long`.

JDK 9 идет в комплекте с методом `Math.multiplyFull(int x, int y)`, возвращающим значение типа `long`. Этот метод очень полезен для получения точного математического произведения двух целых чисел как значения типа `long`:

```
int x = Integer.MAX_VALUE;
int y = Integer.MAX_VALUE;
long z = Math.multiplyFull(x, y); // 4611686014132420609
```

Для справки, JDK 9 также идет в комплекте с методом `Math.multiplyHigh(long x, long y)`. Значение типа `long`, возвращаемое этим методом, представляет собой наиболее значимые 64 бита 128-битного произведения двух 64-битных множителей:

```
long x = Long.MAX_VALUE;
long y = Long.MAX_VALUE;
// 9223372036854775807 * 9223372036854775807 = 4611686018427387903
long z = Math.multiplyHigh(x, y);
```

В контексте функционального стиля потенциальное решение будет опираться на функциональный интерфейс `BinaryOperator` следующим образом (надо просто определить операцию двух операндов одного типа):

```
int x = Integer.MAX_VALUE;
int y = Integer.MAX_VALUE;
BinaryOperator<Integer> operator = Math::multiplyExact;
int z = operator.apply(x, y); // выбрасывает исключение ArithmeticException
```

Во время работы с крупным числом также обратите внимание на класс `BigInteger` (немутуируемые целые числа произвольной точности) и класс `BigDecimal` (немутуируемые знаковые десятичные числа произвольной точности).

38. Совмещенное умножение-сложение (Fused Multiply Add)

Математические вычисления в форме $(a \cdot b) + c$ широко эксплуатируются в матричных умножениях, которые часто используются в высокопроизводительных вычислениях (high-performance computing, HPC), приложениях искусственного интеллекта, машинном обучении, глубоком обучении, нейронных сетях и т. д.

Простейший способ имплементации этого вычисления непосредственно опирается на операторы `*` и `+`, а именно:

```
double x = 49.29d;
double y = -28.58d;
double z = 33.63d;
double q = (x * y) + z;
```

Главная проблема этой имплементации состоит в низкой точности и производительности, обусловленных двумя ошибками округления (одна в операции умножения и другая в операции сложения).

Но благодаря инструкциям Intel AVX для выполнения операций SIMD и среде разработки JDK 9, в которую был добавлен метод `Math.fma()`, это вычисление можно ускорить. Опираясь на метод `Math.fma()`, округление выполняется только один раз с использованием режима округления до ближайшего четного:

```
double fma = Math.fma(x, y, z);
```

Обратите внимание, что это улучшение доступно для современных процессоров Intel, и значит, иметь только среду разработки JDK 9 недостаточно.

39. Компактное форматирование чисел

Начиная с JDK 12, был добавлен новый класс для компактного форматирования чисел — `java.text.CompactNumberFormat`. Главная цель этого класса — расширить существующий в среде Java API форматирования чисел поддержкой локали и сжатия.

Число может быть отформатировано в короткий стиль (например, *1000* становится *1K*) или в длинный стиль (например, *1000* становится *1 thousand*). Эти два стиля были сгруппированы в подклассе `Style` класса перечислений `Enum` в качестве его вариантов `SHORT` и `LONG`.

Помимо конструктора `CompactNumberFormat`, класс `CompactNumberFormat` может создаваться посредством двух статических методов, которые добавлены в класс `NumberFormat`:

- ♦ первым является компактный формат чисел для локали, установленной по умолчанию, с `NumberFormat.Style.SHORT`:

```
public static NumberFormat getCompactNumberInstance()
```

♦ вторым является компактный формат чисел для указанной локали с `NumberFormat.Style`:

```
public static NumberFormat getCompactNumberInstance(  
    Locale locale, NumberFormat.Style formatStyle)
```

Давайте внимательно рассмотрим форматирование и разбор чисел.

Форматирование

По умолчанию число форматируется с использованием `RoundingMode.HALF_EVEN`. Однако мы можем явно задать режим округления посредством `NumberFormat.setRoundingMode()`.

Попытку сжать эту информацию в служебный класс с именем `NumberFormatters` можно осуществить следующим образом:

```
public static String forLocale(Locale locale, double number) {  
    return format(locale, Style.SHORT, null, number);  
}  
  
public static String forLocaleStyle(  
    Locale locale, Style style, double number) {  
  
    return format(locale, style, null, number);  
}  
  
public static String forLocaleStyleRound(  
    Locale locale, Style style, RoundingMode mode, double number) {  
  
    return format(locale, style, mode, number);  
}  
  
private static String format(  
    Locale locale, Style style, RoundingMode mode, double number) {  
  
    if (locale == null || style == null) {  
        return String.valueOf(number); // или использовать формат по умолчанию  
    }  
  
    NumberFormat nf = NumberFormat.getCompactNumberInstance(locale, style);  
  
    if (mode != null) {  
        nf.setRoundingMode(mode);  
    }  
  
    return nf.format(number);  
}
```

Теперь давайте отформатируем числа 1000, 1000000 и 1000000000 с помощью локали США, стиля SHORT и режима округления по умолчанию:

```
// 1K
NumberFormatters.forLocaleStyle(Locale.US, Style.SHORT, 1_000);

// 1M
NumberFormatters.forLocaleStyle(Locale.US, Style.SHORT, 1_000_000);

// 1B
NumberFormatters.forLocaleStyle(Locale.US, Style.SHORT, 1_000_000_000);
```

Мы можем сделать то же самое со стилем LONG:

```
// 1thousand
NumberFormatters.forLocaleStyle(Locale.US, Style.LONG, 1_000);

// 1million
NumberFormatters.forLocaleStyle(Locale.US, Style.LONG, 1_000_000);

// 1billion
NumberFormatters.forLocaleStyle(Locale.US, Style.LONG, 1_000_000_000);
```

Мы также можем использовать итальянскую локаль и стиль SHORT:

```
// 1.000
NumberFormatters.forLocaleStyle(Locale.ITALIAN, Style.SHORT, 1_000);

// 1 Mln
NumberFormatters.forLocaleStyle(Locale.ITALIAN, Style.SHORT, 1_000_000);

// 1 Mld
NumberFormatters.forLocaleStyle(Locale.ITALIAN, Style.SHORT, 1_000_000_000);
```

Наконец, мы также можем применить итальянскую локаль и стиль LONG:

```
// 1 mille
NumberFormatters.forLocaleStyle(Locale.ITALIAN, Style.LONG, 1_000);

// 1 milione
NumberFormatters.forLocaleStyle(Locale.ITALIAN, Style.LONG, 1_000_000);

// 1 miliardo
NumberFormatters.forLocaleStyle(Locale.ITALIAN, Style.LONG, 1_000_000_000);
```

Теперь предположим, что у нас есть два числа: 1200 и 1600.

С точки зрения режима округления, они будут округлены соответственно до 1000 и 2000. Режим округления HALF_EVEN, установленный по умолчанию, округляет число

1200 до 1000 и число 1600 до 2000. Но если мы хотим, чтобы число 1200 стало 2000, а число 1600 стало 1000, то нам нужно настроить режим округления явным образом вот так:

```
// 2000 (2 тысячи)
NumberFormatters.forLocaleStyleRound(
    Locale.US, Style.LONG, RoundingMode.UP, 1_200);

// 1000 (1 тысяча)
NumberFormatters.forLocaleStyleRound(
    Locale.US, Style.LONG, RoundingMode.DOWN, 1_600);
```

Разбор числа

Разбор числа — это обратный форматированию процесс. Мы имеем заданную строку и пытаемся разобрать ее как число. Это можно сделать посредством метода `NumberFormat.parse()`. По умолчанию группирование в процессе разбора не задействуется (например, без группирования результатом разбора строки *5,50 K* будет *5*, а с группированием результатом — *550000*).

Если мы сожмем эту информацию в набор вспомогательных методов, то получим следующий результат:

```
public static Number parseLocale(Locale locale, String number)
    throws ParseException {

    return parse(locale, Style.SHORT, false, number);
}

public static Number parseLocaleStyle(
    Locale locale, Style style, String number) throws ParseException {

    return parse(locale, style, false, number);
}

public static Number parseLocaleStyleRound(
    Locale locale, Style style, boolean grouping, String number)
    throws ParseException {

    return parse(locale, style, grouping, number);
}

private static Number parse(
    Locale locale, Style style, boolean grouping, String number)
    throws ParseException {

    if (locale == null || style == null || number == null) {
        throw new IllegalArgumentException(
            "Locale/style/number cannot be null");
    }
}
```

```
NumberFormat nf = NumberFormat.getCompactNumberInstance(locale, style);
nf.setGroupingUsed(grouping);

return nf.parse(number);
}
```

Давайте выполним разбор *5K* и *5 thousand* в *5000* без явно заданного группирования:

```
// 5000
NumberFormatters.parseLocaleStyle(Locale.US, Style.SHORT, "5K");

// 5000
NumberFormatters.parseLocaleStyle(Locale.US, Style.LONG, "5 thousand");
```

Теперь давайте выполним разбор *5,50 K* и *5,50 thousand* в *550000* с явно заданным группированием:

```
// 550000
NumberFormatters.parseLocaleStyleRound(
    Locale.US, Style.SHORT, true, "5,50K");

// 550000
NumberFormatters.parseLocaleStyleRound(
    Locale.US, Style.LONG, true, "5,50 thousand");
```

Более точную настройку можно получить посредством методов `setCurrency()`, `setParseIntegerOnly()`, `setMaximumIntegerDigits()`, `setMinimumIntegerDigits()`, `setMinimumFractionDigits()` и `setMaximumFractionDigits()`.

Резюме

В этой главе собрана группа наиболее часто встречающихся задач с привлечением строк и чисел. Очевидно, что имеется огромная масса таких задач, и попытка охватить их все выходит далеко за рамки любой книги. Однако осведомленность о том, как решать представленные в этой главе задачи, дает вам прочную основу для самостоятельного решения многих других родственных задач.

Скачайте приложения из этой главы, чтобы увидеть результаты и получить дополнительную информацию.

2

Объекты, немутуируемость и выражения *switch*

Эта глава содержит 18 задач с привлечением объектов, немутуируемости и выражений *switch*. Она начинается с нескольких задач, касающихся работы с ссылками *null*, а затем продолжается задачами, связанными с проверкой индексов, методами *equals()* и *hashCode()*, а также немутуируемостью (например, написанием немутуируемых классов и передачей/возвратом мутуируемых объектов из немутуируемых классов). Последняя часть главы посвящена клонирующим объектам и выражениям *switch* версии JDK 12.

К концу этой главы вы получите основополагающие знания об объектах и их немутуируемости. Более того, вы будете знать, как обращаться с новыми выражениями *switch*. Эти ценные и несравнимые знания имеются в арсенале любого разработчика в среде Java.

Задачи

Используйте следующие задачи для проверки вашего умения программировать объекты, немутуируемость и выражения *switch*. Перед тем как обратиться к решениям и скачать примеры программ, настоятельно рекомендуется изучить каждую задачу.

40. **Проверка ссылок на *null* в функциональном и императивном стилях программирования.** Написать программу, которая выполняет проверки заданных ссылок на *null* в функциональном и императивном стилях программирования.
41. **Проверка ссылок на *null* и выбрасывание собственного исключения *NullPointerException*.** Написать программу, которая выполняет проверку заданных ссылок на *null* и выбрасывает исключение *NullPointerException* со специализированными сообщениями.

42. **Проверка ссылок на null и выбрасывание заданного исключения (например, исключения `IllegalArgumentException`).** Написать программу, которая выполняет проверки заданных ссылок на null и выбрасывает указанное исключение.
43. **Проверка ссылок на null и возврат непустых ссылок, заданных по умолчанию.** Написать программу, которая выполняет проверки заданной ссылки на null, и если она не является null, то возвращает ее; в противном случае возвращает непустую ссылку по умолчанию.
44. **Проверка индекса в интервале от 0 до длины.** Написать программу, которая проверяет, находится ли заданный индекс между 0 (включительно) и заданной длиной (исключительно). Если заданный индекс находится вне интервала [0; заданной длины], то выбросить исключение `IndexOutOfBoundsException`.
45. **Проверка подынтервала в интервале от 0 до длины.** Написать программу, которая проверяет, чтобы заданный подынтервал [заданное начало; заданный конец] находился внутри интервала [0; заданная длина]. Если заданный подынтервал отсутствует внутри интервала [0; заданная длина], то выбросить исключение `IndexOutOfBoundsException`.
46. **Методы `equals()` и `hashCode()`.** Объяснить и проиллюстрировать принцип работы методов `equals()` и `hashCode()` в языке Java.
47. **Немутуируемые объекты в двух словах.** Объяснить и проиллюстрировать, что такое немутуируемый объект в среде Java.
48. **Немутуируемая строка.** Объяснить, почему класс `String` является немутуируемым.
49. **Немутуируемый класс.** Написать программу, представляющую немутуируемый класс.
50. **Передача мутируемых объектов в немутуируемый класс и возврат мутируемых объектов из него.** Написать программу, которая передает мутируемый объект в немутуируемый класс и возвращает его из немутуируемого класса.
51. **Написание немутуируемого класса с помощью шаблона строителя.** Написать программу, представляющую имплементацию шаблона строителя в немутуируемом классе.
52. **Предотвращение плохих данных в неизменяемых объектах.** Написать программу, которая предотвращает плохие данные в немутуируемых объектах.
53. **Клонирование объектов.** Написать программу, которая иллюстрирует методы мелкого и глубокого клонирования.
54. **Переопределение метода `toString()`.** Объяснить и продемонстрировать способы переопределения метода `toString()`.
55. **Выражения `switch`.** Предоставить краткий обзор инструкции и выражений `switch` в JDK 12.

56. **Многочисленные метки вариантов.** Написать фрагмент кода для примера выражения `switch` JDK 12 с многочисленными метками вариантов `case`.
57. **Блоки инструкций.** Написать фрагмент кода с примером выражения `switch` JDK 12, в котором метки вариантов `case` указывают на блок в фигурных скобках.

Решения

В следующих разделах описаны решения перечисленных выше задач. Помните, что обычно не существует одного единственно правильного варианта решения той или иной задачи. Кроме того, имейте в виду, что приведенные здесь объяснения включают только самые интересные и важные детали, необходимые для решения. Вы можете скачать примеры решений, чтобы увидеть дополнительные детали и поэкспериментировать с программами, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

40. Проверка ссылок на *null* в функциональном и императивном стилях программирования

Независимо от стиля программирования — функционального или императивного — проверка ссылок на `null` является распространенным и рекомендуемым техническим приемом, используемым для смягчения последствий возникновения всем известного исключения `NullPointerException`. Этот вид проверки активно используется для аргументов методов с целью обеспечения того, чтобы передаваемые ссылки не становились причиной исключения `NullPointerException` либо непредвиденного поведения.

Например, передача аргумента `List<Integer>` в метод может потребовать как минимум двух проверок на `null`. Во-первых, данный метод должен обеспечивать, чтобы сама ссылка на список не являлась `null`. Во-вторых, в зависимости от того, как используется список, данный метод должен обеспечивать, чтобы список не содержал объектов `null`:

```
List<Integer> numbers
    = Arrays.asList(1, 2, null, 4, null, 16, 7, null);
```

Этот список передается в следующий ниже метод:

```
public static List<Integer> evenIntegers(List<Integer> integers) {
    if (integers == null) {
        return Collections.EMPTY_LIST;
    }

    List<Integer> evens = new ArrayList<>();
    for (Integer nr: integers) {
        if (nr != null && nr % 2 == 0) {
```

```
        evens.add(nr);
    }
}

return evens;
}
```

Обратите внимание, что в приведенном выше фрагменте кода используются классические проверки, основанные на операторах `==` и `!=` (`integers==null, nr!=null`). Начиная с JDK 8, класс `java.util.Objects` содержит два метода, которые оборачивают проверки на `null`, основанные на двух указанных операторах: `object == null` был обернут в метод `Objects.isNull()`, а `object != null` был обернут в метод `Objects.nonNull()`.

На основе этих методов приведенный выше фрагмент кода можно переписать следующим образом:

```
public static List<Integer> evenIntegers(List<Integer> integers) {
    if (Objects.isNull(integers)) {
        return Collections.EMPTY_LIST;
    }

    List<Integer> evens = new ArrayList<>();

    for (Integer nr: integers) {
        if (Objects.nonNull(nr) && nr % 2 == 0) {
            evens.add(nr);
        }
    }

    return evens;
}
```

Теперь исходный код стал чуть выразительнее, но это использование двух указанных методов не является главным. На самом деле оба метода были добавлены для другой цели (в соответствии с примечаниями к API) — для использования в качестве предикатов в исходном коде в функциональном стиле Java 8. Проверки на `null` в исходном коде в функциональном стиле могут выполняться, как показано в следующих ниже примерах:

```
public static int sumIntegers(List<Integer> integers) {
    if (integers == null) {
        throw new IllegalArgumentException("List cannot be null");
    }

    return integers.stream()
        .filter(i -> i != null)
```

```
        .mapToInt(Integer::intValue).sum();
    }

    public static boolean integersContainsNulls(List<Integer> integers) {
        if (integers == null) {
            return false;
        }

        return integers.stream()
            .anyMatch(i -> i == null);
    }
```

Совершенно очевидно, что `i -> i != null` и `i -> i == null` не выражены в едином стиле с окружающим кодом. Давайте заменим эти фрагменты кода на `Objects.nonNull()` и `Objects.isNull()`:

```
    public static int sumIntegers(List<Integer> integers) {
        if (integers == null) {
            throw new IllegalArgumentException("List cannot be null");
        }

        return integers.stream()
            .filter(Objects::nonNull)
            .mapToInt(Integer::intValue).sum();
    }

    public static boolean integersContainsNulls(List<Integer> integers) {
        if (integers == null) {
            return false;
        }

        return integers.stream()
            .anyMatch(Objects::isNull);
    }
```

В качестве альтернативы мы можем использовать методы `Objects.nonNull()` и `Objects.isNull()` также для аргументов:

```
    public static int sumIntegers(List<Integer> integers) {
        if (Objects.isNull(integers)) {
            throw new IllegalArgumentException("List cannot be null");
        }

        return integers.stream()
            .filter(Objects::nonNull)
            .mapToInt(Integer::intValue).sum();
    }
```

```
public static boolean integersContainsNulls(List<Integer> integers) {  
    if (Objects.isNull(integers)) {  
        return false;  
    }  
  
    return integers.stream()  
        .anyMatch(Objects::isNull);  
}
```

Потрясающе! Таким образом, в порядке заключения всякий раз, когда требуются проверки на null, исходный код в функциональном стиле должен опираться на эти два метода, тогда как в исходном коде в императивном стиле это является вопросом предпочтений.

41. Проверка ссылок на *null* и выбрасывание собственного исключения *NullPointerException*

Проверка ссылок на null и выбрасывание исключений `NullPointerException` с собственными сообщениями может выполняться с использованием следующего фрагмента кода (приведенный ниже код делает это четыре раза — дважды в конструкторе и дважды в методе `assignDriver()`):

```
public class Car {  
    private final String name;  
    private final Color color;  
  
    public Car(String name, Color color) {  
        if (name == null) {  
            throw new NullPointerException("Имя автомобиля не может быть null");  
        }  
  
        if (color == null) {  
            throw new NullPointerException("Цвет автомобиля не может быть null");  
        }  
  
        this.name = name;  
        this.color = color;  
    }  
  
    public void assignDriver(String license, Point location) {  
        if (license == null) {  
            throw new NullPointerException("Лицензия не может быть null");  
        }  
    }  
}
```

```
    if (location == null) {  
        throw new NullPointerException("Местоположение не может быть null");  
    }  
}  
}
```

Каким образом, этот код решает задачу путем комбинирования оператора `==` и ручного создания экземпляра класса `NullPointerException`. Начиная с JDK 7, это комбинирование кода было спрятано в статический метод с именем `Objects.requireNonNull()`. Посредством данного метода приведенный выше код можно переписать выразительным образом:

```
public class Car {  
    private final String name;  
    private final Color color;  
  
    public Car(String name, Color color) {  
        this.name = Objects.requireNonNull(name,  
            "Имя автомобиля не может быть null");  
        this.color = Objects.requireNonNull(color,  
            "Цвет автомобиля не может быть null");  
    }  
  
    public void assignDriver(String license, Point location) {  
        Objects.requireNonNull(license, "Лицензия не может быть null");  
        Objects.requireNonNull(location, "Местоположение не может быть null");  
    }  
}
```

Таким образом, если заданная ссылка равна `null`, то метод `Objects.requireNonNull()` выбрасывает исключение `NullPointerException` с заданным сообщением. В противном случае он возвращает проверенную ссылку.

В конструкторах существует типичный подход выбрасывать исключение `NullPointerException`, когда предоставляемые ссылки равны `null`. Но в методах (например, в методе `assignDriver()`) этот подход является спорным. Некоторые разработчики предпочитают возвращать безобидный результат либо выбрасывать исключение `IllegalArgumentException`. Следующая задача — проверка ссылок на `null` и выбрасывание заданного исключения (например, исключения `IllegalArgumentException`) — обращается к подходу на основе исключения `IllegalArgumentException`.

В JDK 7 есть два метода `Objects.requireNonNull()`, один из которых использовался ранее. Другой метод выбрасывает исключение `NullPointerException` с сообщением по умолчанию, как в следующем примере:

```
this.name = Objects.requireNonNull(name);
```

Начиная с JDK 8, есть еще один метод `Objects.requireNonNull()`. Он оборачивает специализированное сообщение `NullPointerException` в функциональный интерфейс `Supplier`. Это означает, что создание сообщения откладывается до тех пор, пока заданная ссылка не станет `null` (и значит, использование оператора `+` для конкатенации частей сообщения больше не является проблемой).

Вот пример:

```
this.name = Objects.requireNonNull(name, ()
    -> "Имя автомобиля не может быть null ... Возьмите имя из " + carsList);
```

Если эта ссылка не равна `null`, сообщение не создается.

42. Проверка ссылок на *null* и выбрасывание заданного исключения

Конечно, одно из решений непосредственно опирается на оператор `==` следующим образом:

```
if (name == null) {
    throw new IllegalArgumentException("Имя не может быть null");
}
```

Эту задачу не получится решить посредством методов `java.util.Objects`, т. к. отсутствует метод `requireNonNullElseThrow()`. Для выбрасывания исключения `IllegalArgumentException` или другого заданного исключения может потребоваться набор методов, показанных на снимке экрана на рис. 2.1.

● <code>requireNonNullElseThrow(T obj, X exception)</code>	T
● <code>requireNonNullElseThrowIAE(T obj, String message)</code>	T
● <code>requireNonNullElseThrowIAE(T obj, Supplier<String> messageSupplier)</code>	T
● <code>requireNotNullElseThrow(T obj, Supplier<? extends X> exceptionSupplier)</code>	T

Рис. 2.1

Давайте сосредоточимся на методах `requireNonNullElseThrowIAE()`. Оба метода выбрасывают исключение `IllegalArgumentException` со специализированным сообщением, указываемым как `String` или как `Supplier` (во избежание создания до тех пор, пока проверка на значение `null` не станет `true`):

```
public static <T> T requireNonNullElseThrowIAE(T obj, String message) {
    if (obj == null) {
        throw new IllegalArgumentException(message);
    }

    return obj;
}

public static <T> T requireNonNullElseThrowIAE(T obj,
    Supplier<String> messageSupplier) {
```



```
if (obj == null) {
    throw new IllegalArgumentException(messageSupplier == null
        ? null : messageSupplier.get());
}

return obj;
}
```

Таким образом, выбрасывать исключения `IllegalArgumentException` можно посредством этих двух методов. Но их недостаточно. Например, коду может потребоваться выбрасывать исключения `IllegalStateException`, `UnsupportedOperationException` и т. д.

Для таких случаев предпочтительны такие методы:

```
public static <T, X extends Throwable> T requireNonNullElseThrow(
    T obj, X exception) throws X {

    if (obj == null) {
        throw exception;
    }

    return obj;
}

public static <T, X extends Throwable> T requireNotNullElseThrow(
    T obj, Supplier<? extends X> exceptionSupplier) throws X {

    if (obj != null) {
        return obj;
    } else {
        throw exceptionSupplier.get();
    }
}
```

Следует подумать о добавлении этих методов во вспомогательный класс `MyObjects`. Эти методы вызываются, как показано в приведенном ниже примере:

```
public Car(String name, Color color) {
    this.name = MyObjects.requireNonNullElseThrow(name,
        new UnsupportedOperationException("Имя не может быть установлено в null"));
    this.color = MyObjects.requireNotNullElseThrow(color, () ->
        new UnsupportedOperationException("Цвет не может быть установлен в null"));
}
```

Далее мы можем следовать этим примерам, чтобы обогатить класс `MyObjects` исключениями другого рода.

43. Проверка ссылок на *null* и возврат непустых ссылок, заданных по умолчанию

Решение этой задачи можно легко обеспечить посредством инструкции `if-else` (или тернарной инструкции), как в следующем ниже примере (альтернативно, `name` и `color` можно объявить как не `final` и инициализировать значениями, установленными по умолчанию, при объявлении):

```
public class Car {
    private final String name;
    private final Color color;
    public Car(String name, Color color) {

        if (name == null) {
            this.name = "Безымянный";
        } else {
            this.name = name;
        }

        if (color == null) {
            this.color = new Color(0, 0, 0);
        } else {
            this.color = color;
        }
    }
}
```

Однако, начиная с JDK 9, приведенный выше фрагмент кода можно упростить двумя методами из класса `Objects`: `requireNonNullElse()` и `requireNonNullElseGet()`. Оба метода берут два аргумента — ссылку для проверки на `null` и непустую ссылку, заданную по умолчанию, для возврата в случае, если проверяемая ссылка равна `null`:

```
public class Car {
    private final String name;
    private final Color color;

    public Car(String name, Color color) {
        this.name = Objects.requireNonNullElse(name, "No name");
        this.color = Objects.requireNonNullElseGet(color,
            () -> new Color(0, 0, 0));
    }
}
```

В приведенном выше примере эти методы используются в конструкторе, но они также могут применяться и в методах.

44. Проверка индекса в интервале от 0 до длины

Для начала давайте рассмотрим простой сценарий, чтобы со стороны взглянуть на эту задачу. Данный сценарий материализуется в следующем простом классе:

```
public class Function {  
    private final int x;  
  
    public Function(int x) {  
        this.x = x;  
    }  
  
    public int xMinusY(int y) {  
        return x - y;  
    }  
  
    public static int oneMinusY(int y) {  
        return 1 - y;  
    }  
}
```

Обратите внимание, что приведенный выше фрагмент кода не накладывает каких-либо интервальных ограничений на аргументы *x* и *y*. Теперь давайте зададим следующие интервалы (так часто делают при использовании математических функций):

- ◆ значение *x* должно быть между 0 (включительно) и 11 (исключительно), чтобы *x* принадлежало [0; 11);
- ◆ в методе `xMinusY()` значение *y* должно находиться между 0 (включительно) и *x* (исключительно), чтобы *y* принадлежало [0; *x*);
- ◆ в методе `oneMinusY()` значение *y* должно быть между 0 (включительно) и 16 (исключительно), чтобы *y* принадлежало [0; 16).

Эти интервалы можно ввести в код посредством инструкций `if` следующим образом:

```
public class Function {  
    private static final int X_UPPER_BOUND = 11;  
    private static final int Y_UPPER_BOUND = 16;  
    private final int x;  
  
    public Function(int x) {  
        if (x < 0 || x >= X_UPPER_BOUND) {  
            throw new IndexOutOfBoundsException("...");  
        }  
        this.x = x;  
    }  
}
```

```
public int xMinusY(int y) {
    if (y < 0 || y >= x) {
        throw new IndexOutOfBoundsException("...");
    }
    return x - y;
}

public static int oneMinusY(int y) {
    if (y < 0 || y >= Y_UPPER_BOUND) {
        throw new IndexOutOfBoundsException("...");
    }
    return 1 - y;
}
}
```

Стоит подумать о замене исключения `IndexOutOfBoundsException` более осмысленным исключением (например, расширить исключение `IndexOutOfBoundsException` и создать собственное исключение наподобие `RangeOutOfBoundsException`).

Начиная с JDK 9, этот исходный код можно переписать с использованием метода `Objects.checkIndex()`, который проверяет, чтобы заданный индекс находился в интервале `[0; длина)`, и возвращает заданный индекс в этом интервале или выбрасывает исключение `IndexOutOfBoundsException`:

```
public class Function {
    private static final int X_UPPER_BOUND = 11;
    private static final int Y_UPPER_BOUND = 16;
    private final int x;

    public Function(int x) {
        this.x = Objects.checkIndex(x, X_UPPER_BOUND);
    }

    public int xMinusY(int y) {
        Objects.checkIndex(y, x);

        return x - y;
    }

    public static int oneMinusY(int y) {
        Objects.checkIndex(y, Y_UPPER_BOUND);

        return 1 - y;
    }
}
```

Например, вызов `oneMinusY()`, как показано в следующем ниже фрагменте кода, в результате выбросит исключение `IndexOutOfBoundsException`, т. к. `y` может принимать значения только в интервале `[0; 16)`:

```
int result = Function.oneMinusY(20);
```

Теперь давайте пойдем дальше и проверим подынтервал в интервале от 0 до заданной длины.

45. Проверка подынтервала в интервале от 0 до длины

Давайте проследим ту же самую процедуру из предыдущей задачи. Итак, на этот раз класс `Function` будет выглядеть следующим образом:

```
public class Function {
    private final int n;

    public Function(int n) {
        this.n = n;
    }

    public int yMinusX(int x, int y) {
        return y - x;
    }
}
```

Обратите внимание, что приведенный выше фрагмент кода не накладывает каких-либо интервальных ограничений на аргументы `x`, `y` и `n`. Теперь давайте введем следующие интервалы:

- ♦ значение `n` должно быть между 0 (включительно) и 101 (исключительно), чтобы `n` принадлежало `[0; 101)`;
- ♦ в методе `yMinusX()` интервал, ограниченный значениями `x` и `y`, — `[x; y)` должен быть подынтервалом `[0; n]`.

Эти интервалы могут быть выражены в коде посредством инструкций `if` следующим образом:

```
public class Function {
    private static final int N_UPPER_BOUND = 101;
    private final int n;

    public Function(int n) {
        if (n < 0 || n >= N_UPPER_BOUND) {
            throw new IndexOutOfBoundsException("...");
        }
        this.n = n;
    }
}
```

```

public int yMinusX(int x, int y) {
    if (x < 0 || x > y || y >= n) {
        throw new IndexOutOfBoundsException("...");
    }
    return y - x;
}
}

```

Опираясь на предыдущую задачу, условие для `n` может быть заменено методом `Objects.checkIndex()`. Более того, класс `JDK 9 Objects` идет в комплекте с методом `checkFromToIndex(int start, int end, int length)`, который проверяет, чтобы заданный подынтервал [заданное начало; заданный конец] находился внутри интервала [0; заданная длина]. Таким образом, этот метод можно применить к методу `yMinusX()`, чтобы проверить, что интервал, ограниченный значениями `x` и `y`, `[x; y)` является подынтервалом интервала `[0; n]`:

```

public class Function {
    private static final int N_UPPER_BOUND = 101;
    private final int n;

    public Function(int n) {
        this.n = Objects.checkIndex(n, N_UPPER_BOUND);
    }

    public int yMinusX(int x, int y) {
        Objects.checkFromToIndex(x, y, n);

        return y - x;
    }
}

```

Например, следующий ниже тест приведет к исключению `IndexOutOfBoundsException`, т. к. `x` больше `y`:

```

Function f = new Function(50);
int r = f.yMinusX(30, 20);

```



Помимо этого метода, класс `Objects` идет с еще одним методом с именем `checkFromIndexSize(int start, int size, int length)`. Этот метод проверяет, что подынтервал [заданное начало; заданное начало + заданный размер] находится внутри интервала [0; заданная длина].

46. Методы `equals()` и `hashCode()`

Методы `equals()` и `hashCode()` определены в классе `java.lang.Object`. Поскольку класс `Object` является надклассом всех объектов Java, оба метода доступны для всех объектов. Их главная цель состоит в том, чтобы обеспечить простое, эффективное и

робастное решение для сравнения объектов и выяснять, являются ли они эквивалентными. Без этих методов и их контрактов решение опирается на большие и громоздкие инструкции `if`, предназначенные для сравнения каждого поля объекта.

Когда эти методы не переопределены, язык Java будет использовать их имплементации по умолчанию. К сожалению, такая имплементация на самом деле не служит цели выяснения того, что два объекта имеют одинаковые значения. По умолчанию метод `equals()` проверяет идентичность. Другими словами, он считает, что два объекта равны, если и только если они представлены одним и тем же адресом в памяти (одинаковыми ссылками на объекты), тогда как метод `hashCode()` возвращает целочисленное представление адреса объекта в памяти. Эта функциональность является нативной и называется *идентификационным хеш-кодом*.

Например, предположим, что существует следующий класс:

```
public class Player {
    private int id;
    private String name;

    public Player(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

Затем создадим два экземпляра этого класса, содержащих одну и ту же информацию, и сравним их на эквивалентность:

```
Player p1 = new Player(1, "Рафаэль Надаль");
```

```
Player p2 = new Player(1, "Рафаэль Надаль");
```

```
System.out.println(p1.equals(p2)); // false
```

```
System.out.println("хеш-код p1: " + p1.hashCode()); // 1809787067
```

```
System.out.println("хеш-код p2: " + p2.hashCode()); // 157627094
```



Не используйте оператор `==` для проверки эквивалентности объектов (избегайте `if (p1 == p2)`). Оператор `==` сравнивает на предмет того, что ссылки двух объектов указывают на один и тот же объект, тогда как метод `equals()` сравнивает значения объектов (нас интересует именно это).

В качестве общего правила, если две переменные содержат одинаковую ссылку, то они являются *идентичными*, но если они ссылаются на одинаковое значение, то они являются эквивалентными. То, что подразумевается под "*одинаковым значением*", определяется методом `equals()`.

Для нас `p1` и `p2` являются эквивалентными, но обратите внимание, что метод `equals()` вернул `false` (экземпляры `p1` и `p2` имеют точно такие же значения полей, но они хранятся по разным адресам памяти). Это означает, что опираться на имплементацию метода `equals()` по умолчанию недопустимо. Решение состоит в том,

чтобы переопределить этот метод, и для этого важно знать о контракте метода `equals()`, который накладывает следующие условия:

- ◆ **рефлексивность** — объект эквивалентен самому себе, т. е. `p1.equals(p1)` должно возвращать `true`;
- ◆ **симметрия** — `p1.equals(p2)` должно возвращать тот же результат (`true/false`), что и `p2.equals(p1)`;
- ◆ **транзитивность** — если `p1.equals(p2)` и `p2.equals(p3)`, то `p1.equals(p3)`;
- ◆ **согласованность** — два эквивалентных объекта должны оставаться эквивалентными все время, пока один из них не будет изменен;
- ◆ **Null возвращает false** — все объекты не должны быть равны `null`.

Итак, для соблюдения этого контракта метод `equals()` класса `Player` можно переопределить следующим образом:

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }

    if (obj == null) {
        return false;
    }

    if (getClass() != obj.getClass()) {
        return false;
    }

    final Player other = (Player) obj;

    if (this.id != other.id) {
        return false;
    }

    if (!Objects.equals(this.name, other.name)) {
        return false;
    }

    return true;
}
```

Теперь снова выполним тест на эквивалентность (на этот раз `p1` эквивалентен `p2`):

```
System.out.println(p1.equals(p2));    // true
```


ОК, пока все хорошо! Теперь давайте добавим вот эти два экземпляра класса `Player` в коллекцию. Например, добавим их в коллекцию `HashSet` (хеш-множество Java, которое не допускает повторов):

```
Set<Player> players = new HashSet<>();
players.add(p1);
players.add(p2);
```

Давайте проверим размер этой коллекции `HashSet` и выясним, содержит ли она `p1`:

```
System.out.println("хеш-код p1: " + p1.hashCode());           // 1809787067
System.out.println("хеш-код p2: " + p2.hashCode());           // 157627094
System.out.println("Размер множества: " + players.size());    // 2
System.out.println("Множество содержит Рафаэля Надаля: "
    + players.contains(new Player(1, "Рафаэль Надаль")));      // false
```

В соответствии с предыдущей имплементацией метода `equals()`, `p1` и `p2` эквивалентны; поэтому размер коллекции `HashSet` должен быть 1, а не 2. Более того, она должна содержать Рафаэля Надаля. Итак, что же произошло?

Дело в том, что общий ответ зиждется на том, как сконструирована среда Java. Легко интуитивно понять, что метод `equals()` не является быстрым; следовательно, поисковые запросы будут сталкиваться с проблемами производительности, когда требуется значительное число сравнений эквивалентности. Например, метод имеет серьезный недостаток в случае поиска конкретных значений в коллекциях (например, коллекциях `HashSet`, `HashMap` и `HashTable`), т. к. он будет требовать большого числа сравнений равенства.

Основываясь на этом утверждении, разработчики среды Java попытались сократить число сравнений эквивалентности, добавив корзины. *Корзина* — это контейнер на основе хеша, который группирует равные объекты. Это означает, что эквивалентные объекты должны возвращать одинаковый хеш-код, тогда как неэквивалентные объекты должны возвращать разные хеш-коды (если два неэквивалентных объекта имеют одинаковый хеш-код, то такая ситуация называется *хеш-коллизией*, и объекты будут находиться в одинаковой корзине). Таким образом, язык Java сравнивает хеш-коды, и только если они являются одинаковыми для двух разных ссылок на объекты (а не для одинаковых ссылок на объекты), то программа идет дальше и вызывает метод `equals()`. В сущности, такой подход ускоряет поиск в коллекциях.

Но что же произошло в нашем случае? Давайте посмотрим на это в пошаговом режиме:

- ◆ когда будет создан экземпляр `p1`, Java назначит ему хеш-код на основе адреса `p1` в памяти;
- ◆ когда `p1` добавляется в коллекцию `HashSet`, Java свяжет новую корзину с хеш-кодом `p1`;
- ◆ когда будет создан экземпляр `p2`, Java назначит ему хеш-код на основе адреса `p2` в памяти;

- ◆ когда p2 добавляется в коллекцию `HashSet`, Java свяжет новую корзину с хеш-кодом p2 (когда это происходит, похоже, что коллекция `HashSet` работает не так, как ожидалось, и она допускает повторы);
- ◆ когда выполняется инструкция `players.contains(new Player(1, "Рафаэль Надаль"))`, новый игрок p3 создается с новым хеш-кодом, основанным на адресе p3 в памяти;
- ◆ итак, в рамках метода `contains()` проверка p1 и p3 и, соответственно, p2 и p3, на эквивалентность предусматривает проверку их хеш-кодов, а т. к. хеш-код p1 отличается от хеш-кода p3, а хеш-код p2 отличается от хеш-кода p3, то сравнения прекращаются без вычисления метода `equals()`, и значит, коллекция `HashSet` не содержит объекта (p3).

Для того чтобы можно было вернуться в нужное русло, код должен также переопределить метод `hashCode()`. Контракт метода `hashCode()` накладывает следующие ограничения:

- ◆ два эквивалентных объекта, соответствующие требованиям метода `equals()`, должны возвращать одинаковый хеш-код;
- ◆ два объекта с одинаковым хеш-кодом не обязательно являются эквивалентными;
- ◆ пока объект остается неизменным, метод `hashCode()` должен возвращать одинаковое значение.

В качестве общего правила, для того чтобы соблюдать контракты методов `equals()` и `hashCode()`, следуйте двум золотым правилам:

- ◆ когда метод `equals()` переопределяется, метод `hashCode()` должен быть переопределен тоже, и наоборот;
- ◆ используйте для обоих методов одинаковые идентифицирующие атрибуты в одном и том же порядке.

Для класса `Player` метод `hashCode()` можно переопределить следующим образом:

```
@Override
public int hashCode() {
    int hash = 7;
    hash = 79 * hash + this.id;
    hash = 79 * hash + Objects.hashCode(this.name);

    return hash;
}
```

Теперь давайте выполним еще один тест (на этот раз он работает, как и ожидалось):

```
System.out.println("хеш-код p1: " + p1.hashCode());           // -322171805
System.out.println("хеш-код p2: " + p2.hashCode());           // -322171805
System.out.println("Размер множества: " + players.size());    // 2
System.out.println("Множество содержит Рафаэля Надаля: "
    + players.contains(new Player(1, "Рафаэль Надаль")));      // true
```

Теперь перечислим несколько часто встречающихся ошибок работы с методами `equals()` и `hashCode()`:

- ◆ вы переопределяете метод `equals()` и забываете переопределить метод `hashCode()`, или наоборот (следует переопределять оба или ни одного);
- ◆ для сравнения значений объектов используется оператор `==` вместо функции метода `equals()`;
- ◆ в методе `equals()` вы опускаете одно или несколько из следующих значений:
 - начать с добавления самопроверки (`if (this == obj)...`);
 - поскольку ни один экземпляр не должен быть равен `null`, продолжить, добавив *проверку на null* (`if (obj == null)...`);
 - обеспечить, чтобы экземпляр был ожидаемым (применить метод `getClass()` или `instanceof`);
 - наконец, после этих крайних случаев добавить сравнения полей;
 - вы нарушаете симметрию метода `equals()` посредством наследования. Допустим, что у нас есть класс `A` и класс `B`, расширяющий `A` и добавляющий новое поле. Класс `B` переопределяет имплементацию метода `equals()`, унаследованную от `A`, и эта имплементация добавляется в новое поле. Применение метода `instanceof` покажет, что `b.equals(a)` будет возвращать `false` (как и ожидалось), но `a.equals(b)` будет возвращать `true` (что не ожидалось), поэтому симметрия нарушается. Сравнение *срезов* не сработает, т. к. оно нарушает транзитивность и рефлексивность. Исправление проблемы подразумевает использование метода `getClass()` вместо метода `instanceof` (благодаря методу `getClass()` экземпляры типа и его подтипов не могут быть эквивалентными), либо вместо наследования лучше применить композицию, как в приложении, прилагаемом к этой книге (`P46_ViolateEqualsViaSymmetry`);
 - из метода `hashCode()` вместо уникального хеш-кода вы возвращаете константу для каждого объекта.

Начиная с JDK 7, класс `Objects` идет в комплекте с несколькими помощниками для работы с объектной эквивалентностью и хеш-кодами, как показано ниже:

- ◆ `Objects.equals(Object a, Object b)` проверяет эквивалентность объекта `a` объекту `b`;
- ◆ `Objects.deepEquals(Object a, Object b)` полезен для проверки эквивалентности двух объектов (если это массивы, то проверка выполняется посредством метода `Arrays.deepEquals()`);
- ◆ `Objects.hash(Object ... values)` генерирует хеш-код для последовательности входных значений.



Следует обеспечивать соблюдение контрактов Java SE методами `equals()` и `hashCode()` посредством библиотеки `EqualsVerifier` (<https://mvnrepository.com/artifact/nl.jqno.equalsverifier/equalsverifier>).

Для генерирования методов `hashCode()` и `equals()` из полей вашего объекта следует использовать библиотеку `Lombok` (<https://projectlombok.org>). Но обратите внимание на частный случай комбинирования библиотеки `Lombok` с сущностями JPA.

47. Немутуируемые объекты в двух словах

Немутуируемый объект — это объект, который не может быть изменен (его состояние фиксировано) после его создания.

В языке Java действуют следующие правила:

- ◆ примитивные типы являются немутуируемыми;
- ◆ всем известный класс `String` среды Java является немутуируемым (другие классы тоже являются немутуируемыми, например `Pattern` и `LocalDate`);
- ◆ массивы не являются немутуируемыми;
- ◆ коллекции могут быть мутуируемыми, немодифицируемыми или немутуируемыми.

Немодифицируемая коллекция не является автоматически немутуируемой. Все зависит от того, какие объекты хранятся в коллекции. Если хранящиеся объекты являются мутуируемыми, то коллекция мутуируемая и немодифицируемая. Но если хранящиеся объекты являются немутуируемыми, то коллекция фактически немутуируемая.

Немутуируемые объекты полезны в конкурентных (многопоточных)¹ приложениях и потоках. Поскольку немутуируемые объекты не могут быть изменены, они не подвержены проблемам конкурентности и не рискуют быть поврежденными или несогласованными.

Одна из главных проблем использования немутуируемых объектов связана со штрафами за создание новых объектов вместо управления состоянием мутуируемого объекта. Но имейте в виду, что немутуируемые объекты имеют преимущество, состоящее в особом обращении во время сбора мусора. Более того, они не подвержены проблемам конкурентности и исключают код, необходимый для управления состоянием мутуируемых объектов. Код, необходимый для управления состоянием мутуируемых объектов, обычно работает медленнее, чем создание новых объектов.

Рассмотрение следующих далее задач позволит нам углубиться в немутуируемость объектов в среде Java.

¹ Обратитесь к разд. "Комментарии переводчика" в начале книги, где даны пояснения терминов "конкурентный" (concurrency) и "нить исполнения" (thread). — Прим. перев.

48. Немутлируемая строка

Каждый язык программирования имеет свой способ представления цепочек символов, или строк. Как примитивные типы, строки являются частью предопределенных типов, и они используются почти во всех видах приложений Java.

В среде Java строки не представлены примитивным типом, таким как типы `int`, `long` и `float`. Они представлены ссылочным типом с именем `String`. Практически любое приложение Java использует строки, например, метод `main()` приложения Java получает в качестве аргумента массив значений типа `String`.

Всеобщая известность типа `String` и его широкий спектр применения означают, что мы должны знать его в деталях. Помимо того что разработчики должны знать, как объявлять и манипулировать строками (например, инвертировать или писать с заглавной буквы), они должны понимать, почему этот тип был разработан особым или отличающимся образом. Точнее, почему строка является немутлируемой? Или, возможно, этот вопрос найдет большее понимание, если его сформулировать вот так: каковы плюсы и минусы немутлируемости типа `String`?

Достоинства немутлируемости строк

Давайте рассмотрим некоторые плюсы немутлируемости строк в следующих далее разделах.

Пул строковых констант или пул кэшированных строк

Одна из причин в пользу немутлируемости строк представлена **пулом строковых констант** (`string constant pool`, SCP) или пулом кэшированных строк. Для того чтобы понять это положение, немного углубимся в то, как работает класс `String` внутри.

Пул строковых констант — это специальная область в памяти (не обычная динамическая память, или куча), используемая для хранения строковых литералов. Допустим, что у нас есть следующие три строковые переменные:

```
String x = "book";  
String y = "book";  
String z = "book";
```

Сколько объектов класса `String` было создано? Заманчиво сказать, что три, но на самом деле Java создает только один строковый объект со значением `"book"` (рис. 2.2). Идея заключается в том, что все, что находится между кавычками, рассматривается как строковый литерал, и Java хранит строковые литералы в этой специальной области памяти, именуемой пулом строковых констант, следуя алгоритму, подобному приведенному ниже (этот алгоритм называется **интернированием строк** — от англ. *string interning*):

- ♦ при создании строкового литерала (например, `String x = "book"`) язык Java проверяет пул строковых констант, чтобы увидеть, что этот строковый литерал существует;

- ◆ если строковый литерал не найден в пуле строковых констант, то в этом пуле создается новый строковый объект для строкового литерала и соответствующая переменная `x` будет указывать на него;
- ◆ если строковый литерал в пуле строковых констант найден (например, `String y = "book"`, `String z = "book"`), то новая переменная будет указывать на этот объект класса `String` (в сущности, все переменные, имеющие одинаковые значения, будут указывать на одинаковый объект класса `String`):

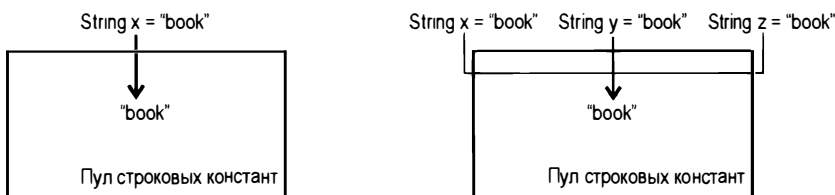


Рис. 2.2

Но переменная `x` должна иметь значение `"cook"`, а не `"book"`, поэтому давайте заменим `"b"` на `"c"`:

```
x.replace("b", "c");
```

В то время как переменная `x` должна стать `"cook"`, переменные `y` и `z` должны остаться неизменными. Такое поведение обеспечивается немутуируемостью. Java создаст новый объект и выполнит над ним изменения, показанные на рис. 2.3.

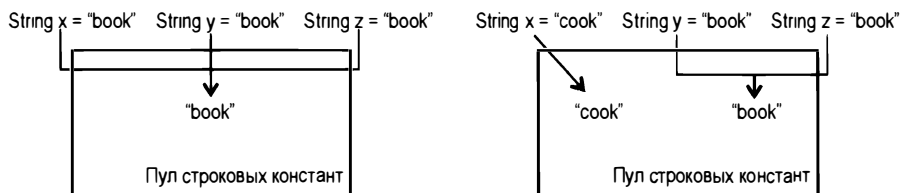


Рис. 2.3

Таким образом, немутуируемость строк позволяет кэшировать строковые литералы, что дает возможность приложениям использовать большое число строковых литералов с минимальным влиянием на динамическую память (кучу) и сборщик мусора. В мутуируемом контексте модификация строкового литерала может привести к повреждению переменных.



Не следует создавать строку как `String x = new String("book")`. Это не строковый литерал; это экземпляр класса `String` (построенный посредством конструктора), который будет помещен в памяти в обычную кучу вместо пула строковых констант. Строка, созданная в обычной куче, может указывать на пул строковых констант посредством явного вызова метода `String.intern()` как `x.intern()`.

Безопасность

Еще одним преимуществом немутуируемости строк является их безопасность. Обычно в виде строк представляется и передается масса конфиденциальной информации (пользовательские имена, пароли, URL-адреса, порты, базы данных, сетевые соединения, параметры, свойства и т. д.). Имея эту информацию немутуируемой, программный код становится безопасным для широкого спектра угроз безопасности (например, модифицирование ссылок нечаянно или преднамеренно).

Нитебезопасность

Представьте себе приложение, использующее тысячи мутуируемых объектов `String` и работающее с нитебезопасным кодом. К счастью, в этом случае наш воображаемый сценарий не станет реальностью благодаря немутуируемости. Любой немутуируемый объект по своей природе является нитебезопасным. Это означает, что строки могут использоваться совместно и управляться более одной нитью исполнения без риска повреждения и несогласованности.

Кэширование хеш-кодов

В разд. "46. Методы `equals()` и `hashCode()`" ранее в этой главе обсуждались методы `equals()` и `hashCode()`. Хеш-коды должны вычисляться всякий раз, когда они участвуют в хешировании конкретных действий (например, при поиске элемента в коллекции). Поскольку класс `String` является немутуируемым, каждая строка имеет немутуируемый хеш-код, который можно кэшировать и использовать повторно, поскольку он не может изменяться после создания строки. Это означает, что хеш-коды строк можно использовать из кэша вместо того, чтобы их пересчитывать при каждом использовании. Так, коллекция `HashMap` хеширует свои ключи для разных операций (например, `put()`, `get()`), и если эти ключи имеют тип `String`, то хеш-коды будут использоваться повторно из кэша вместо их пересчета.

Загрузка классов

Типичный подход к загрузке класса в память основан на вызове метода `Class.forName(String className)`. Обратите внимание на аргумент типа `String`, представляющий имя класса. Благодаря немутуируемости строк имя класса не может изменяться в процессе загрузки. Однако если бы тип `String` был мутуируемым, то представьте себе загрузку `class A` (например, `Class.forName("A")`), и в процессе загрузки его имя было бы изменено на `BadA`. Теперь объекты `BadA` могли бы наделать много чего плохого!

Недостатки немутуируемости строк

Рассмотрим некоторые минусы немутуируемости строк в следующих далее разделах.

Строка не может расширяться

Немутуируемый класс следует объявлять `final` во избежание расширения. Однако разработчикам часто требуется возможность расширения класса `String`, чтобы добавить туда больше способностей, и это ограничение можно считать недостатком немутуируемости.

Тем не менее разработчики могут писать служебные классы (например, класс `StringUtils` в библиотеке `Apache Commons Lang`, класс `StringUtils` в прикладном каркасе `Spring` и класс `strings` в библиотеке `Guava`) для предоставления добавленных способностей и просто передавать строки в качестве аргументов методам этих классов.

Чувствительные данные в памяти в течение продолжительного времени

Чувствительные данные в строках (например, пароли) могут храниться в памяти (в пул строковых констант) в течение продолжительного времени. Будучи кэшем, пул строковых констант использует преимущества специального обращения со стороны сборщика мусора. Точнее, пул строковых констант не посещается сборщиком мусора с той же частотой (циклическостью), что и другие участки памяти. В результате такой специальной обработки чувствительные данные хранятся в пуле строковых констант в течение продолжительного времени и могут быть подвержены нежелательному использованию.

Во избежание этого потенциального недостатка рекомендуется хранить чувствительные данные (например, пароли) в массиве `char[]` вместо значения типа `String`.

Ошибка `OutOfMemoryError`

Пул строковых констант представляет собой малый участок памяти по сравнению с другими и может быть заполнен довольно быстро. Хранение слишком большого числа строковых литералов в пуле строковых констант приводит к ошибке исчерпания памяти `OutOfMemoryError`.

Является ли класс `String` полностью немутуируемым?

Дело в том, что за кулисами для хранения каждого символа строкового значения класс `String` использует `private final char[]`. С помощью API рефлексии среды `Java` в `JDK 8` следующий ниже фрагмент кода выполнит модификацию этого массива `char[]` (тот же код в `JDK 11` выбросит исключение `java.lang.ClassCastException`):

```
String user = "guest";
System.out.println("Тип пользователя: " + user);
```

```
Class<String> type = String.class;
Field field = type.getDeclaredField("value");
field.setAccessible(true);
```

```
char[] chars = (char[]) field.get(user);
```



```
chars[0] = 'a';
chars[1] = 'd';
chars[2] = 'm';
chars[3] = 'i';
chars[4] = 'n';
```

```
System.out.println("Тип пользователя: " + user);
```

Таким образом, в JDK 8 тип `String` *фактически* является немутуируемым, но не *полностью*.

49. Немутуируемый класс

Немутуируемый класс должен удовлетворять нескольким требованиям, а именно:

- ◆ класс должен быть помечен как `final`, чтобы подавить расширяемость (другие классы не могут расширять этот класс; следовательно, они не могут переопределять его методы);
- ◆ все поля должны быть объявлены приватными (`private`) и финальными (`final`) (они не видны в других классах и инициализируются только один раз в конструкторе этого класса);
- ◆ класс должен содержать параметризованный публичный (`public`) конструктор (или приватный (`private`) конструктор и фабричные методы для создания экземпляров), который инициализирует поля;
- ◆ класс должен предоставлять геттеры, т. е. методы чтения, для полей;
- ◆ класс не должен выставлять наружу сеттеры, т. е. методы записи/мутации.

Например, следующий ниже класс `Point` является немутуируемым, т. к. он успешно проходит приведенный выше контрольный список:

```
public final class Point {
    private final double x;
    private final double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }
}
```

Если немутуируемый класс должен манипулировать мутуируемыми объектами, то нужно рассмотреть следующие далее задачи.

50. Передача мутуируемых объектов в немутуируемый класс и возврат мутуируемых объектов из него

Передача мутуируемых объектов в немутуируемый класс может привести к нарушению немутуируемости. Рассмотрим следующий мутуируемый класс:

```
public class Radius {
    private int start;
    private int end;

    public int getStart() {
        return start;
    }

    public void setStart(int start) {
        this.start = start;
    }

    public int getEnd() {
        return end;
    }

    public void setEnd(int end) {
        this.end = end;
    }
}
```

Затем передадим экземпляр этого класса немутуируемому классу с именем `Point`. На первый взгляд класс `Point` можно написать следующим образом:

```
public final class Point {
    private final double x;
    private final double y;
    private final Radius radius;

    public Point(double x, double y, Radius radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    public double getX() {
        return x;
    }
}
```

```
public double getY() {
    return y;
}

public Radius getRadius() {
    return radius;
}
}
```

Является ли этот класс по-прежнему немутуируемым? Ответ — нет. Класс `Point` больше не является немутуируемым, т. к. его состояние может быть изменено, как в следующем примере:

```
Radius r = new Radius();
r.setStart(0);
r.setEnd(120);
```

```
Point p = new Point(1.23, 4.12, r);
```

```
System.out.println("Radius start: " + p.getRadius().getStart()); // 0
r.setStart(5);
System.out.println("Radius start: " + p.getRadius().getStart()); // 5
```

Обратите внимание, что вызовы метода `p.getRadius().getStart()` вернули два разных результата; следовательно, состояние `p` было изменено, поэтому класс `Point` больше не является немутуируемым. Решением этой проблемы является клонирование объекта `Radius` и сохранение клона в качестве поля класса `Point`:

```
public final class Point {
    private final double x;
    private final double y;
    private final Radius radius;

    public Point(double x, double y, Radius radius) {
        this.x = x;
        this.y = y;

        Radius clone = new Radius();
        clone.setStart(radius.getStart());
        clone.setEnd(radius.getEnd());

        this.radius = clone;
    }

    public double getX() {
        return x;
    }
}
```

```

public double getY() {
    return y;
}

public Radius getRadius() {
    return radius;
}
}

```

На этот раз уровень немутуируемости класса `Point` увеличился (вызов метода `r.setStart(5)` не повлияет на поле `radius`, т. к. это поле — клон объекта `r`). Но класс `Point` не является полностью немутуируемым, потому что есть еще одна проблема, которую нужно решить — возвращение мутуируемых объектов из немутуируемого класса может нарушить немутуируемость. Взгляните на следующий фрагмент кода, который нарушает немутуируемость класса `Point`:

```

Radius r = new Radius();
r.setStart(0);
r.setEnd(120);

Point p = new Point(1.23, 4.12, r);

System.out.println("Начало радиуса: " + p.getRadius().getStart()); // 0
p.getRadius().setStart(5);
System.out.println("Начало радиуса: " + p.getRadius().getStart()); // 5

```

Опять-таки, вызовы метода `p.getRadius().getStart()` вернули два разных результата; следовательно, состояние `p` было изменено. Решение состоит в модифицировании метода `getRadius()` так, чтобы он возвращал клон поля `radius`, следующим образом:

```

...
public Radius getRadius() {
    Radius clone = new Radius();
    clone.setStart(this.radius.getStart());
    clone.setEnd(this.radius.getEnd());

    return clone;
}
...

```

Теперь класс `Point` снова является немутуируемым. Задача решена!



Прежде чем выбрать технический прием/инструмент клонирования, в некоторых случаях рекомендуется не торопиться и проанализировать/изучить разные возможности, доступные в Java и в сторонних библиотеках (например, обратиться к разд. "53. Клонирование объектов" далее в этой главе). В случае мелких копий приведенный выше технический прием может быть надлежащим вариантом выбора, но для глубоких копий, исходному коду, возможно, придется опираться на

другие подходы, такие как конструктор копирования, интерфейс `Cloneable`, или внешние библиотеки (например, библиотеку `Apache Commons Lang` и ее класс `ObjectUtils`, сериализацию JSON с помощью библиотек `Gson` или `Jackson` либо любые другие).

51. Написание немутуруемого класса с помощью шаблона строителя

Когда класс (немутуруемый или мутуруемый) имеет слишком много полей, он требует конструктора с большим числом аргументов. Если некоторые из этих полей являются обязательными, а другие необязательными, то этому классу потребуется несколько конструкторов, чтобы охватить все возможные комбинации. Он становится громоздким для разработчика и для пользователя класса. Именно здесь на помощь приходит шаблон строителя.

Согласно книге "банды четырех" (*Gang of Four, GoF*) о шаблонах архитектурного дизайна, шаблон строителя отделяет построение многосложного объекта от его представления, благодаря чему один и тот же процесс строительства может создавать разные представления.

Шаблон строителя может быть имплементирован как отдельный класс или как внутренний статический (`static`) класс. Давайте сосредоточимся на втором случае. Класс `User` содержит три обязательных поля (`nickname`, `password` и `created`) и три необязательных поля (`email`, `firstname` и `lastname`).

Теперь немутуруемый класс `User`, опирающийся на шаблон строителя, будет выглядеть следующим образом:

```
public final class User {
    private final String nickname;
    private final String password;
    private final String firstname;
    private final String lastname;
    private final String email;
    private final Date created;

    private User(UserBuilder builder) {
        this.nickname = builder.nickname;
        this.password = builder.password;
        this.created = builder.created;
        this.firstname = builder.firstname;
        this.lastname = builder.lastname;
        this.email = builder.email;
    }

    public static UserBuilder getBuilder(
        String nickname, String password) {
        return new User.UserBuilder(nickname, password);
    }
}
```

```
public static final class UserBuilder {
    private final String nickname;
    private final String password;
    private final Date created;
    private String email;
    private String firstname;
    private String lastname;

    public UserBuilder(String nickname, String password) {
        this.nickname = nickname;
        this.password = password;
        this.created = new Date();
    }

    public UserBuilder firstName(String firstname) {
        this.firstname = firstname;
        return this;
    }

    public UserBuilder lastName(String lastname) {
        this.lastname = lastname;
        return this;
    }

    public UserBuilder email(String email) {
        this.email = email;
        return this;
    }

    public User build() {
        return new User(this);
    }
}

public String getNickname() {
    return nickname;
}

public String getPassword() {
    return password;
}

public String getFirstname() {
    return firstname;
}
```

```
public String getLastname() {
    return lastname;
}

public String getEmail() {
    return email;
}

public Date getCreated() {
    return new Date(created.getTime());
}
}
```

Вот несколько примеров использования:

```
import static modern.challenge.User.getBuilder;
...
// пользователем с ником и паролем
User user1 = getBuilder("marin21", "hju9887h").build();

// пользователем с ником, паролем и электронной почтой
User user2 = getBuilder("ionk", "44fef22")
    .email("ion@gmail.com")
    .build();

// пользователем с ником, паролем, электронной почтой, именем и фамилией
User user3 = getBuilder("monika", "klooi0988")
    .email("monika@gmail.com")
    .firstName("Monika")
    .lastName("Ghuenter")
    .build();
```

52. Предотвращение плохих данных в немутуируемых объектах

Плохие данные — это любые данные, которые оказывают негативное влияние на немутуируемый объект (например, поврежденные данные). Скорее всего, эти данные поступают от пользователей или из внешних источников данных, которые не находятся под нашим непосредственным контролем. В таких случаях плохие данные могут ударить по немутуируемому объекту, и хуже всего то, что нет никакой возможности от этого избавиться. Немутуируемый объект не может быть изменен после создания; следовательно, плохие данные будут жить счастливо все время, пока живет объект.

Решение этой проблемы заключается в проверке всех данных, входящих в немутуируемый объект, на соответствие всеобъемлющему набору ограничений.

Существуют разные способы выполнения указанной проверки, от специализированной проверки до встроенных решений. Проверка может выполняться вне или внутри класса немутуируемых объектов в зависимости от дизайна приложения. Например, если немутуируемый объект построен с помощью шаблона строителя, то проверка может быть выполнена в строительном классе.

JSR 380 представляет собой спецификацию API Java для проверки компонентов (бинов) объектной модели (Java SE/EE). Эта спецификация может использоваться для проверки с помощью аннотаций. Валидатор Hibernate Validator является эталонной имплементацией API проверки, и его можно легко предоставить в качестве Maven-зависимости в файл `pom.xml` (сверьтесь с исходным кодом, прилагаемым к этой книге).

Далее мы опираемся на выделенные аннотации, чтобы обеспечить необходимые ограничения (например, `@NotNull`, `@Min`, `@Max`, `@Size` и `@Email`). В приведенном ниже примере ограничения добавляются в строительный класс следующим образом:

```
...
public static final class UserBuilder {
    @NotNull(message = "не может быть null")
    @Size(min = 3, max = 20, message = " должен быть от 3 до 20 символов ")
    private final String nickname;

    @NotNull(message = "не может быть null")
    @Size(min = 6, max = 50, message = " должен быть от 6 до 50 символов ")
    private final String password;

    @Size(min = 3, max = 20, message = " должен быть от 3 до 20 символов ")
    private String firstname;

    @Size(min = 3, max = 20, message = "должен быть от 3 до 20 символов")
    private String lastname;

    @Email(message = "должен быть допустимым")
    private String email;

    private final Date created;

    public UserBuilder(String nickname, String password) {
        this.nickname = nickname;
        this.password = password;
        this.created = new Date();
    }
    ...
}
```


Наконец, процесс проверки запускается из кода через API валидатора (это необходимо только в Java SE). Если данные, входящие в строительный класс, недопустимы, то немутулируемый объект не создается (не вызывается метод `build()`):

```
User user;
Validator validator
    = Validation.buildDefaultValidatorFactory().getValidator();

User.UserBuilder userBuilder
    = new User.UserBuilder("monika", "klooi0988")
        .email("monika@gmail.com")
        .firstName("Monika").lastName("Gunther");

final Set<ConstraintViolation<User.UserBuilder>> violations
    = validator.validate(userBuilder);
if (violations.isEmpty()) {
    user = userBuilder.build();
    System.out.println("User успешно создан на: " + user.getCreated());
} else {
    printConstraintViolations("Нарушения UserBuilder: ", violations);
}
```

Таким образом, плохие данные не могут коснуться немутулируемого объекта. Если нет строительного класса, то ограничения могут быть добавлены непосредственно на уровне поля в немутулируемом объекте. Приведенное выше решение просто показывает потенциальные нарушения в консоли, но, в зависимости от ситуации, данное решение может выполнять разные действия (например, выбрасывать конкретные исключения).

53. Клонирование объектов

Операция клонирования объектов выполняется не каждый день, но важно проводить ее правильно. В общих чертах, клонирование объектов относится к созданию копий объектов. Существует два главных вида копий — *мелкие* копии (копировать как можно меньше) и *глубокие* копии (копировать все).

Допустим, что у нас есть следующий класс:

```
public class Point {
    private double x;
    private double y;

    public Point() {}

    public Point(double x, double y) {
        this.x = x;
```

```
    this.y = y;
}
// геттеры и сеттеры
}
```

Таким образом, у нас есть точка вида (x, y) , отображенная в классе. А теперь давайте выполним клонирование.

Клонирование ручное

Быстрый подход состоит в добавлении метода, который копирует текущую точку `Point` в новую точку `Point` вручную (эта копия является мелкой):

```
public Point clonePoint() {
    Point point = new Point();
    point.setX(this.x);
    point.setY(this.y);
    return point;
}
```

Здесь код довольно простой: создать новый экземпляр класса `Point` и заполнить его поля полями текущей точки `Point`. Возвращаемая точка `Point` является мелкой копией (поскольку `Point` не зависит от других объектов, глубокая копия будет точно такой же) текущей точки `Point`:

```
Point point = new Point(...);
Point clone = point.clonePoint();
```

Клонирование методом `clone()`

Класс `Object` содержит метод `clone()`. Этот метод полезен для создания мелких копий (его можно использовать и для глубоких копий). Для того чтобы его использовать, класс должен выполнить следующие шаги:

- ◆ имплементировать интерфейс `Cloneable` (если этот интерфейс не имплементирован, будет выброшено исключение `CloneNotSupportedException`);
- ◆ переопределить метод `clone()` (`Object.clone()` является защищенным, т. е. `protected`);
- ◆ вызвать `super.clone()`.

Интерфейс `Cloneable` не содержит никаких методов. Это просто сигнал для JVM, что этот объект можно клонировать. После того как этот интерфейс имплементирован, код должен переопределить метод `Object.clone()`. Это необходимо потому, что метод `Object.clone()` защищенный, и, чтобы вызвать его через `super`, код должен переопределить указанный метод. Это становится серьезным недостатком, если метод `clone()` добавляется в дочерний класс, т. к. все надклассы должны определить метод `clone()` во избежание сбоя в цепочке вызовов `super.clone()`.

Более того, `Object.clone()` не опирается на вызов конструктора, поэтому разработчик не может управлять построением объекта:

```
public class Point implements Cloneable {
    private double x;
    private double y;

    public Point() {}

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public Point clone() throws CloneNotSupportedException {
        return (Point) super.clone();
    }

    // геттеры и сеттеры
}
```

Клон может создаваться следующим образом:

```
Point point = new Point(...);
Point clone = point.clone();
```

Клонирование посредством конструктора

Этот технический прием клонирования требует, чтобы вы обогатили класс конструктором, который берет один аргумент, представляющий экземпляр класса, который будет использоваться для создания клона.

Давайте посмотрим на него в коде:

```
public class Point {
    private double x;
    private double y;

    public Point() {}

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public Point(Point another) {
        this.x = another.x;
    }
}
```

```
    this.y = another.y;
}

// геттеры и сеттеры
}
```

Клон может создаваться следующим образом:

```
Point point = new Point(...);
Point clone = new Point(point);
```

Клонирование посредством библиотеки *Cloning*

Глубокое копирование необходимо, когда объект зависит от другого объекта. Выполнение глубокого копирования означает копирование объекта, включая его цепочку зависимостей. Например, допустим, что `Point` имеет поле типа `Radius`:

```
public class Radius {
    private int start;
    private int end;

    // геттеры и сеттеры
}

public class Point {
    private double x;
    private double y;
    private Radius radius;

    public Point(double x, double y, Radius radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    // геттеры и сеттеры
}
```

Выполнение мелкой копии точки `Point` создаст копию `x` и `y`, но не создаст копию объекта `Radius`. Это означает, что изменения, влияющие на объект `Radius`, будут также отражены в клоне. Самое время для глубокой копии.

Громоздкое решение будет включать переделку ранее представленных приемов мелкого копирования с целью поддержки глубокого копирования. К счастью, есть несколько решений, которые можно применить прямо из коробки, и одним из них является библиотека клонирования `CloningCloning` (<https://github.com/kostaskougios/cloning>):

```
import com.rits.cloning.Cloner;
...
```

```
Point point = new Point(...);
Cloner cloner = new Cloner();
Point clone = cloner.deepClone(point);
```

Приведенный выше исходный код не требует пояснений. Обратите внимание, что библиотека Cloning идет в придачу с несколькими другими средствами, как видно на снимке экрана на рис. 2.4.

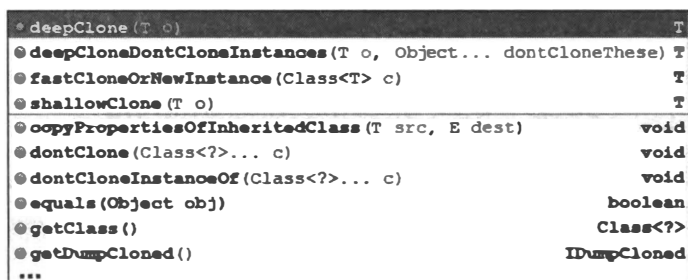


Рис. 2.4

Клонирование посредством сериализации

Этот технический прием требует наличия сериализуемых объектов (impleментирующих `java.io.Serializable`). В сущности, объект сериализуется (методом `writeObject()`) и десериализуется (методом `readObject()`) в новый объект. Вспомогательный метод, способный это выполнить, выглядит следующим образом:

```
private static <T> T cloneThroughSerialization(T t) {
    try {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(baos);
        oos.writeObject(t);

        ByteArrayInputStream bais
            = new ByteArrayInputStream(baos.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bais);

        return (T) ois.readObject();
    } catch (IOException | ClassNotFoundException ex) {
        // запротоколировать исключение
        return t;
    }
}
```

Таким образом, объект сериализуется в поток `ObjectOutputStream` и десериализуется в поток `ObjectInputStream`. Клонировать объект с помощью этого метода можно следующим образом:

```
Point point = new Point(...);
Point clone = cloneThroughSerialization(point);
```

^{*} Встроенное решение, основанное на сериализации, предоставляется библиотекой **Apache Commons Lang** посредством класса `SerializationUtils`. Среди своих методов этот класс имеет метод `clone()`, который можно использовать следующим образом:

```
Point point = new Point(...);
Point clone = SerializationUtils.clone(point);
```

Клонирование посредством JSON

Почти любая библиотека JSON в Java может выполнять сериализацию любого обыкновенного объекта Java (plain old Java object, POJO) без необходимости дополнительного конфигурирования/отображения. Наличие библиотеки JSON в проекте (а многие проекты ее имеют) может спасти нас от добавления дополнительной библиотеки с целью обеспечения глубокого клонирования. В общих чертах решение задачи может использовать существующую библиотеку JSON для получения того же эффекта.

Ниже приведен пример использования библиотеки Gson:

```
private static <T> T cloneThroughJson(T t) {
    Gson gson = new Gson();
    String json = gson.toJson(t);

    return (T) gson.fromJson(json, t.getClass());
}
```

```
Point point = new Point(...);
Point clone = cloneThroughJson(point);
```

В дополнение к этому всегда есть возможность написать собственную библиотеку, специально посвященную клонированию объектов.

54. Переопределение метода `toString()`

Метод `toString()` определен в `java.lang.Object`, и JDK поставляется с его имплементацией по умолчанию. Эта имплементация автоматически используется для всех объектов, которые являются предметом методов `print()`, `println()`, `printf()`, отладки во время разработки, журналирования/протоколирования, информационных сообщений в исключениях и т. д.

К сожалению, строковое представление объекта, возвращаемого имплементацией по умолчанию, является не очень информативным. Например, рассмотрим следующий класс `User`:

```
public class User {
    private final String nickname;
    private final String password;
    private final String firstname;
    private final String lastname;
```

```
private final String email;
private final Date created;

// конструктор и геттеры опущены для краткости
}
```

Теперь давайте создадим экземпляр этого класса и напечатаем его в консоли:

```
User user = new User("sparg21", "kkd454ffc",
    "Leopold", "Mark", "mark1@yahoo.com");
```

```
System.out.println(user);
```

Результат этого метода `println()` будет выглядеть примерно так, как показано на рис. 2.5.

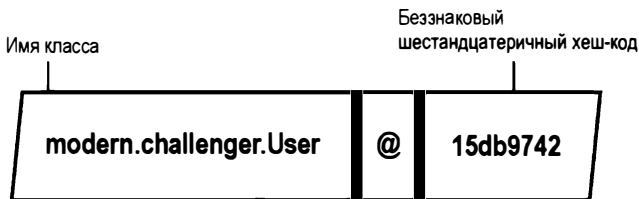


Рис. 2.5

Решение задачи во избежание результата, показанного на рис. 2.5, состоит в переопределении метода `toString()`. Например, давайте переопределим его так, чтобы предоставлять сведения о пользователе:

```
@Override
public String toString() {
    return "User{" + "nickname=" + nickname + ", password=" + password
        + ", firstname=" + firstname + ", lastname=" + lastname
        + ", email=" + email + ", created=" + created + '>';
}
```

На этот раз метод `println()` покажет следующий результат:

```
User {
    nickname = sparg21, password = kkd454ffc,
    firstname = Leopold, lastname = Mark,
    email = mark1@yahoo.com, created = Fri Feb 22 10: 49: 32 EET 2019
}
```

Он гораздо информативнее, чем предыдущий.

Но помните, что метод `toString()` автоматически вызывается для различных целей. Например, вывод в журнал операций может быть таким:

```
logger.log(Level.INFO, "Это потрясающий пользователь: {0}", user);
```

Здесь пароль пользователя попадет в журнал, и это может представлять проблему. Выставление наружу чувствительных журнальных данных, таких как пароли, учет-

ные записи и секретные IP-адреса, в приложении, безусловно, является плохой практикой.

Поэтому обратите особое внимание на тщательный отбор информации, которая входит в метод `toString()`, т. к. эта информация может оказаться в местах, где может быть использована злонамеренно. В нашем случае пароль не должен быть частью метода `toString()`:

```
@Override
public String toString() {
    return "User{" + "nickname=" + nickname
        + ", firstname=" + firstname + ", lastname=" + lastname
        + ", email=" + email + ", created=" + created + '}';
}
```

Обычно метод `toString()` генерируется посредством IDE. Поэтому обратите внимание на то, какие поля вы выбираете, прежде чем IDE создаст для вас исходный код.

55. Выражения *switch*

Прежде чем мы сделаем краткий обзор выражений `switch`, предусмотренных в JDK 12, давайте рассмотрим типичный традиционный пример инструкции `switch`, завернутой в метод:

```
private static Player createPlayer(PlayerTypes playerType) {
    switch (playerType) {
        case TENNIS:
            return new TennisPlayer();
        case FOOTBALL:
            return new FootballPlayer();
        case SNOOKER:
            return new SnookerPlayer();
        case UNKNOWN:
            throw new UnknownPlayerException("Тип игрока неизвестен");
        default:
            throw new IllegalArgumentException(
                "Недопустимый тип игрока: " + playerType);
    }
}
```

Если мы забудем о варианте `default`, то этот код компилироваться не будет.

Очевидно, что приведенный выше пример является приемлемым. В сценарии для наихудшего случая мы можем добавить мнимую переменную (например, `player`), несколько загромождающих инструкций `break` и не получить никаких жалоб, если вариант `default` отсутствует. Итак, следующий ниже код представляет собой традиционную чрезвычайно уродливую инструкцию `switch`:

```
private static Player createPlayerSwitch(PlayerTypes playerType) {
    Player player = null;
```



```
switch (playerType) {
    case TENNIS:
        player = new TennisPlayer();
        break;
    case FOOTBALL:
        player = new FootballPlayer();
        break;
    case SNOOKER:
        player = new SnookerPlayer();
        break;
    case UNKNOWN:
        throw new UnknownPlayerException("Тип игрока неизвестен");
    default:
        throw new IllegalArgumentException(
            "Недопустимый тип игрока: " + playerType);
}
return player;
}
```

Если мы забудем о варианте `default`, то никаких претензий со стороны компилятора не будет. В этом случае пропущенный вариант `default` может в результате дать игрока, равного `null`.

Однако, начиная с JDK 12, мы можем опираться на *выражения* `switch`. До JDK 12 ключевое слово `switch` представляло *инструкцию*, т.е. синтаксическую конструкцию, предназначенную для управления потоком логики программы (например, как инструкция `if`) без представления результата. С другой стороны, выражение вычисляет результат. Поэтому выражение `switch` может иметь результат.

Приведенная выше инструкция `switch` может быть написана в стиле JDK 12 следующим образом:

```
private static Player createPlayer(PlayerTypes playerType) {
    return switch (playerType) {
        case TENNIS ->
            new TennisPlayer();
        case FOOTBALL ->
            new FootballPlayer();
        case SNOOKER ->
            new SnookerPlayer();
        case UNKNOWN ->
            throw new UnknownPlayerException("Тип игрока неизвестен");
        // вариант default не является обязательным
        default ->
            throw new IllegalArgumentException(
                "Недопустимый тип игрока: " + playerType);
    };
}
```

На этот раз вариант `default` не обязателен. Мы можем его пропустить.

Выражение `switch` в версии JDK 12 является достаточно умным для того, чтобы сигнализировать, если `switch` не охватывает все возможные входные значения. Это очень полезно в случае вариантов перечисления `Enum` в Java. Выражение `switch` в версии JDK 12 может обнаруживать охват всех вариантов перечисления и не заставляет использовать бесполезное значение `default`, если охвачены не все варианты. Например, если мы удалим значение `default` и добавим в перечисление `PlayerTypes` (как подкласс класса перечислений `Enum`) новый элемент (например, `GOLF`), то компилятор уведомит нас об этом в сообщении, как на рис. 2.6 (в NetBeans).

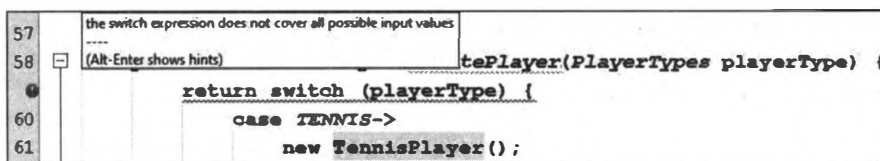


Рис. 2.6

Обратите внимание, что между меткой и исполнением мы заменили двоеточие стрелкой (синтаксис в стиле лямбда-выражений). Главная роль этой стрелки заключается в предотвращении проскакивания далее, т. е. будет исполнен блок кода только справа от нее. Использовать инструкцию `break` нет необходимости.

Не думайте, что стрелка превращает инструкцию `switch` в выражение `switch`. Выражение `switch` также можно использовать с двоеточием и инструкцией `break`, как показано ниже:

```
private static Player createPlayer(PlayerTypes playerType) {
    return switch (playerType) {
        case TENNIS:
            break new TennisPlayer();
        case FOOTBALL:
            break new FootballPlayer();
        case SNOOKER:
            break new SnookerPlayer();
        case UNKNOWN:
            throw new UnknownPlayerException("Тип игрока неизвестен");
        // вариант default не является обязательным
        default:
            throw new IllegalArgumentException(
                "Недопустимый тип игрока: " + playerType);
    };
}
```



В нашем примере выражение `switch` написано поверх типа `enum`, но выражение `switch` JDK 12 также можно использовать поверх типов `int`, `Integer`, `short`, `Short`, `byte`, `Byte`, `char`, `Character` и `String`.

Обратите внимание, что JDK 12 предоставляет выражения `switch` в качестве предварительного функционального средства. Это означает, что оно подвержено изменениям в нескольких следующих релизах, и его необходимо разблокировать с помощью аргумента командной строки `--enable-preview` при компиляции и выполнении.

56. Многочисленные метки вариантов

До JDK 12 инструкция `switch` допускала только одну метку для каждого варианта `case`. Начиная с выражений `switch`, вариант `case` может иметь несколько меток, разделенных запятой. Взгляните на следующий ниже метод, который иллюстрирует несколько меток для варианта `case`:

```
private static SportType
    fetchSportTypeByPlayerType(PlayerTypes playerType) {

    return switch (playerType) {
        case TENNIS, GOLF, SNOOKER ->
            new Individual();
        case FOOTBALL, VOLLEY ->
            new Team();
    };
}
```

Таким образом, если мы передадим в этот метод `TENNIS`, `GOLF` или `SNOOKER`, то он вернет экземпляр класса `Individual`. Если мы передадим `FOOTBALL` или `VOLLEY`, то он вернет экземпляр класса `Team`.

57. Блоки инструкций

Стрелка метки может указывать на одну инструкцию (как в примерах из предыдущих двух задач) или на блок в фигурных скобках. Это очень похоже на лямбда-блоки. Взгляните на следующее ниже решение:

```
private static Player createPlayer(PlayerTypes playerType) {
    return switch (playerType) {
        case TENNIS -> {
            System.out.println("Создается TennisPlayer ...");
            break new TennisPlayer();
        }
        case FOOTBALL -> {
            System.out.println("Создается FootballPlayer ...");
            break new FootballPlayer();
        }
    }
}
```

```
case SNOOKER -> {
    System.out.println("Создается SnookerPlayer ...");
    break new SnookerPlayer();
}
default ->
    throw new IllegalArgumentException(
        "Недопустимый тип игрока: " + playerType);
};
}
```



Обратите внимание, что мы выходим из блока в фигурных скобках посредством инструкции `break`, а не с использованием инструкции `return`. Другими словами, мы можем вернуться (с помощью инструкции `return`) изнутри инструкции `switch`, однако мы не можем вернуться из выражения.

Резюме

Вот и все! Данная глава познакомила вас с несколькими задачами с участием объектов, немутуируемости и выражений `switch`. Задачи, которые охватывали объекты и немутуируемость, представляют собой основополагающие понятия программирования. В отличие от них задачи, охватывающие выражения `switch`, были посвящены введению новых средств JDK 12, затрагивающих эту тему.

Вы можете скачать приложения из этой главы, чтобы увидеть результаты и получить дополнительную информацию, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

3

Работа с датой и временем

Эта глава содержит 20 задач с привлечением даты и времени. Их цель — охватить широкий круг тем (конвертирование, форматирование, сложение, вычитание, определение периодов/продолжительностей, вычисление дат и времени и т. д.) посредством классов `Date`, `Calendar`, `LocalDate`, `LocalTime`, `LocalDateTime`, `ZoneDateTime`, `OffsetDateTime`, `OffsetTime`, `Instant` и т. д.

К концу этой главы у вас не будет проблем с формированием даты и времени, отвечающих потребностям вашего приложения. Основополагающие задачи, представленные в этой главе, очень полезны для получения более широкой картины, касающейся API даты/времени, и будут действовать как элементы пазла, которые необходимо собирать вместе, чтобы разруливать сложные задачи с привлечением даты и времени.

Задачи

Используйте следующие задачи для проверки вашего умения программировать дату и время. Перед тем как обратиться к решениям и скачать примеры программ, настоятельно рекомендуется изучить каждую задачу.

58. **Конвертирование строки в дату и время.** Написать программу, которая иллюстрирует конвертацию из строки в дату-время и обратно.
59. **Форматирование даты и времени.** Объяснить форматный шаблон для даты и времени.
60. **Получение текущих даты и времени без времени или даты.** Написать программу, которая извлекает текущую дату без времени или даты.
61. **Класс `LocalDateTime` из `LocalDate` и `LocalTime`.** Написать программу, которая строит экземпляр класса `LocalDateTime` из объектов `LocalDate` и `LocalTime`. Она объединяет дату и время в одном объекте `LocalDateTime`.
62. **Машинное время посредством класса `Instant`.** Объяснить и привести пример API `Instant`.

63. **Определение временного периода с использованием значений на основе даты (Period) и значений на основе времени (Duration).** Объяснить и привести пример использования API Period и Duration.
64. **Получение единиц даты и времени.** Написать программу, которая извлекает единицы даты и времени (например, извлекает из даты год, месяц, минуту и т. д.) из объекта, представляющего дату-время.
65. **Прибавление к дате и времени и вычитание из них.** Написать программу, которая прибавляет (и вычитает) количество времени (например, годы, дни или минуты) к объекту даты-времени (например, прибавляет час к дате, вычитает 2 дня из LocalDateTime и т. д.).
66. **Получение всех часовых поясов в UTC и GMT.** Написать программу, которая показывает все имеющиеся часовые пояса в UTC и GMT.
67. **Получение локальных даты и времени во всех имеющихся часовых поясах.** Написать программу, которая показывает местное время во всех имеющихся часовых поясах.
68. **Вывод на экран информации о дате и времени полета.** Написать программу, которая показывает информацию о запланированном времени полета, равном 15 часам 30 минутам, а именно время перелета из Перта (Австралия) в Бухарест (Румыния).
69. **Конвертирование временной метки UNIX в дату и время.** Написать программу, которая конвертирует временную метку UNIX в `java.util.Date` и `java.time.LocalDateTime`.
70. **Отыскание первого/последнего дня месяца.** Написать программу, которая отыскивает первый/последний день месяца посредством класса `TemporalAdjusters` JDK 8.
71. **Определение/извлечение поясных смещений.** Написать программу, которая раскрывает разные технические приемы определения и извлечения поясных смещений.
72. **Конвертирование из даты в время и наоборот.** Написать программу, которая конвертирует дату во время и наоборот, используя `LocalDate`, `LocalDateTime` и т. д.
73. **Обход интервала дат в цикле.** Написать программу, которая обходит интервал заданных дат в цикле день за днем (с шагом в один день).
74. **Вычисление возраста.** Написать программу, которая вычисляет возраст человека.
75. **Начало и конец дня.** Написать программу, которая возвращает время начала и конца дня.
76. **Разница между двумя датами.** Написать программу, которая вычисляет временной промежуток в днях между двумя датами.
77. **Имплементация шахматных часов.** Написать программу, которая имплементирует шахматные часы.

Решения

В следующих разделах описаны решения перечисленных выше задач. Помните, что обычно не существует одного единственно правильного варианта решения той или иной задачи. Кроме того, имейте в виду, что приведенные здесь объяснения включают только самые интересные и важные детали, необходимые для решения. Вы можете скачать примеры решений, чтобы увидеть дополнительные детали и поэкспериментировать с программами, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

58. Конвертирование строки в дату и время

Конвертирование или разбор строки в дату и время выполняется посредством набора методов `parse()`. Конвертирование даты и времени в строку осуществляется с помощью методов `toString()` или `format()`.

До JDK 8

До JDK 8 типичное решение этой задачи опиралось на главное расширение абстрактного класса `DateFormat`, именуемого `SimpleDateFormat` (этот класс не является нитебезопасным). В исходном коде, прилагаемом к данной книге, приведено несколько примеров использования этого класса.

Начиная с JDK 8

Начиная с JDK 8, класс `SimpleDateFormat` может быть заменен новым классом — `DateTimeFormatter`. Он является немутурируемым (и, следовательно, нитебезопасным) и используется для печати и разбора объектов даты-времени. Этот класс поддерживает все, начиная со стандартных форматтеров (представленных в виде констант, как локальная дата `ISO`, `2011-12-03`, `iso_local_date`) и заканчивая определяемыми пользователем форматтерами (опираясь на набор символов для написания индивидуальных форматных шаблонов).

Кроме того, помимо класса `Date`, JDK 8 идет в комплекте с несколькими новыми классами, которые предназначены для работы с датой и временем. Некоторые из этих классов показаны в следующем ниже списке (они также называются темпоральными, поскольку имплементируют интерфейс `Temporal`):

- ◆ `LocalDate` (дата без часового пояса в календарной системе ISO-8601);
- ◆ `LocalTime` (время без часового пояса в календарной системе ISO-8601);
- ◆ `LocalDateTime` (дата-время без часового пояса в календарной системе ISO-8601);
- ◆ `ZonedDateTime` (дата-время с часовым поясом в календарной системе ISO-8601);
- ◆ `OffsetDateTime` (дата-время со смещением от UTC/GMT в календарной системе ISO-8601);
- ◆ `OffsetTime` (время со смещением от UTC/GMT в календарной системе ISO-8601).

Для того чтобы можно было конвертировать значение типа `String` в `LocalDate` посредством предопределенного форматтера, оно должно совпадать с шаблоном `DateTimeFormatter.ISO_LOCAL_DATE`, например `2020-06-01`. `LocalDate` предусматривает метод `parse()`, который можно использовать следующим образом:

```
// 06 - это месяц, 01 - это день
```

```
LocalDate localDate = LocalDate.parse("2020-06-01");
```

Схожим образом, в случае `LocalTime` строка должна совпадать с шаблоном `DateTimeFormatter.ISO_LOCAL_TIME`; например, `10:15:30`, как показано в следующем ниже фрагменте кода:

```
LocalTime localTime = LocalTime.parse("12:23:44");
```

В случае `LocalDateTime` строка должна совпадать с шаблоном `DateTimeFormatter.ISO_LOCAL_DATE_TIME`; например, `2020-06-01T11:20:15`, как показано в следующем ниже фрагменте кода:

```
LocalDateTime localDateTime = LocalDateTime.parse("2020-06-01T11:20:15");
```

И в случае `ZonedDateTime` строка должна совпадать с шаблоном `DateTimeFormatter.ISO_ZONED_DATE_TIME`; например, `2020-06-01T10:15:30+09:00[Asia/Tokyo]`, как показано в следующем ниже фрагменте кода:

```
ZonedDateTime zonedDateTime
```

```
= ZonedDateTime.parse("2020-06-01T10:15:30+09:00[Asia/Tokyo]");
```

В случае `OffsetDateTime` строка должна совпадать с шаблоном `DateTimeFormatter.ISO_OFFSET_DATE_TIME`; например, `2007-12-03T10:15:30+01:00`, как показано в следующем ниже фрагменте кода:

```
OffsetDateTime offsetDateTime
```

```
= OffsetDateTime.parse("2007-12-03T10:15:30+01:00");
```

Наконец, в случае `OffsetTime` строка должна совпадать с шаблоном `DateTimeFormatter.ISO_OFFSET_TIME`; например, `10:15:30+01:00`, как показано в следующем ниже фрагменте кода:

```
OffsetTime offsetTime = OffsetTime.parse("10:15:30+01:00");
```

Если строка не совпадает ни с одним из предопределенных форматтеров, самое время определить собственный форматтер посредством специализированного форматного шаблона; например, строка `01.06.2020` представляет дату, которая нуждается в форматтере, определяемом пользователем следующим образом:

```
DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern("dd.MM.yyyy");
```

```
LocalDate localDateFormatted = LocalDate.parse("01.06.2020", dateFormatter);
```

Однако для такой строки, как `12|23|44`, требуется форматтер, определяемый пользователем следующим образом:

```
DateTimeFormatter timeFormatter
```

```
= DateTimeFormatter.ofPattern("HH|mm|ss");
```

```
LocalTime localTimeFormatted
```

```
= LocalTime.parse("12|23|44", timeFormatter);
```


Такая строка, как 01.06.2020, 11:20:15, требует форматтера, определяемого пользователем следующим образом:

```
DateTimeFormatter dateFormatter
    = DateTimeFormatter.ofPattern("dd.MM.yyyy, HH:mm:ss");
LocalDateTime localDateTimeFormatted
    = LocalDateTime.parse("01.06.2020, 11:20:15", dateFormatter);
```

Такая строка, как 01.06.2020, 11:20:15+09:00 [Asia/Tokyo], требует форматтера, определяемого пользователем следующим образом:

```
DateTimeFormatter zonedDateTimeFormatter
    = DateTimeFormatter.ofPattern("dd.MM.yyyy, HH:mm:ssXXXXXX '['VV']'");
ZonedDateTime zonedDateTimeFormatted
    = ZonedDateTime.parse("01.06.2020, 11:20:15+09:00 [Asia/Tokyo]",
        zonedDateTimeFormatter);
```

Такая строка, как 2007.12.03, 10:15:30, +01:00, требует форматтера, определяемого пользователем следующим образом:

```
DateTimeFormatter offsetDateTimeFormatter
    = DateTimeFormatter.ofPattern("yyyy.MM.dd, HH:mm:ss, XXXXX");
OffsetDateTime offsetDateTimeFormatted
    = OffsetDateTime.parse("2007.12.03, 10:15:30, +01:00",
        offsetDateTimeFormatter);
```

Наконец, такая строка, как 10 15 30 +01:00, требует форматтера, определяемого пользователем следующим образом:

```
DateTimeFormatter offsetTimeFormatter
    = DateTimeFormatter.ofPattern("HH mm ss XXXXX");
OffsetTime offsetTimeFormatted
    = OffsetTime.parse("10 15 30 +01:00", offsetTimeFormatter);
```



Каждый метод `ofPattern()` из приведенных выше примеров также поддерживает `Locale`.

Конвертировать из `LocalDate`, `LocalDateTime` или `ZonedDateTime` в `String` можно по крайней мере двумя способами.

♦ Опереться на метод `LocalDate.toString()`, `LocalDateTime.toString()` или `ZonedDateTime.toString()` (автоматически или явно). Обратите внимание, что метод `toString()` всегда будет печатать дату посредством соответствующего predefinedного форматтера:

```
// 2020-06-01 в результате даст ISO_LOCAL_DATE, 2020-06-01
String localDateAsString = localDate.toString();
```

```
// 01.06.2020 в результате даст ISO_LOCAL_DATE, 2020-06-01
String localDateAsString = localDateFormatted.toString();
```

```
// 2020-06-01T11:20:15 в результате даст
// ISO_LOCAL_DATE_TIME, 2020-06-01T11:20:15
String localDateTimeAsString = localDateTime.toString();

// 01.06.2020, 11:20:15 results in
// ISO_LOCAL_DATE_TIME, 2020-06-01T11:20:15
String localDateTimeAsString = localDateTimeFormatted.toString();

// 2020-06-01T10:15:30+09:00[Asia/Tokyo]
// в результате даст ISO_ZONED_DATE_TIME,
// 2020-06-01T11:20:15+09:00[Asia/Tokyo]
String zonedDateTimeAsString = zonedDateTime.toString();

// 01.06.2020, 11:20:15+09:00 [Asia/Tokyo]
// в результате даст ISO_ZONED_DATE_TIME,
// 2020-06-01T11:20:15+09:00[Asia/Tokyo]
String zonedDateTimeAsString = zonedDateTimeFormatted.toString();
```

- ◆ **Опереться на метод `DateTimeFormatter.format()`. Обратите внимание, что метод `DateTimeFormatter.format()` всегда будет печатать дату-время посредством указанного форматтера (по умолчанию часовой пояс будет равен `null`), как показано ниже:**

```
// 01.06.2020
String localDateAsFormattedString
    = dateFormatter.format(localDateFormatted);

// 01.06.2020, 11:20:15
String localDateTimeAsFormattedString
    = dateTimeFormatter.format(localDateTimeFormatted);

// 01.06.2020, 11:20:15+09:00 [Asia/Tokyo]
String zonedDateTimeAsFormattedString
    = zonedDateTimeFormatted.format(zonedDateTimeFormatted);
```

Явный часовой пояс добавляется следующим образом:

```
DateTimeFormatter zonedDateTimeFormatter
    = DateTimeFormatter.ofPattern("dd.MM.yyyy, HH:mm:ssXXXXX '['VV']'")
        .withZone(ZoneId.of("Europe/Paris"));

ZonedDateTime zonedDateTimeFormatted
    = ZonedDateTime.parse("01.06.2020, 11:20:15+09:00 [Asia/Tokyo]",
        zonedDateTimeFormatter);
```

На этот раз строка представляет дату-время в часовом поясе Европа/Париж:

```
// 01.06.2020, 04:20:15+02:00 [Europe/Paris]
String zonedDateTimeAsFormattedString
    = zonedDateTimeFormatted.format(zonedDateTimeFormatter);
```

59. Форматирование даты и времени

Предыдущая задача содержит некоторые варианты форматирования даты и времени посредством методов `SimpleDateFormat.format()` и `DateTimeFormatter.format()`. Для того чтобы определить *форматные шаблоны*, разработчик должен знать их синтаксис. Другими словами, разработчик должен знать набор символов, используемых API даты-времени Java для распознавания допустимого форматного шаблона.

Большинство символов характерны для форматтеров `SimpleDateFormat` (до JDK 8) и `DateTimeFormatter` (начиная с JDK 8). В табл. 3.1 перечислены наиболее распространенные символы, полный список имеется в документации JDK.

Таблица 3.1

Буква	Описание	Представление	Пример
y	Год	год	1994; 94
M	Месяц года	число/текст	7; 07; Jul; July; J
W	Неделя месяца	число	4
E	День недели	текст	Tue; Tuesday; T
d	День месяца	число	15
H	Час дня	число	22
m	Минута часа	число	34
s	Секунда минуты	число	55
S	Доля секунды	число	345
z	Часовой	имя-пояса	Pacific Standard Time; PST
Z	Смещение пояса	смещение-пояса	-0800
V	ID часового пояса (JDK 8)	ID пояса	America/Los_Angeles; Z; -08:30

Несколько примеров форматных шаблонов приведено в табл. 3.2.

Таблица 3.2

Шаблон	Пример
yyyy-MM-dd	2019-02-24
MM-dd-yyyy	02-24-2019
MMM-dd-yyyy	Feb-24-2019
dd-MM-yy	24-02-19
dd.MM.yyyy	24.02.2019
yyyy-MM-dd HH:mm:ss	2019-02-24 11:26:26

Таблица 3.2 (окончание)

Шаблон	Пример
yyyy-MM-dd HH:mm:ssSSS	2019-02-24 11:36:32743
yyyy-MM-dd HH:mm:ssZ	2019-02-24 11:40:35+0200
yyyy-MM-dd HH:mm:ss z	2019-02-24 11:45:03 EET
E MMM yyyy HH:mm:ss.SSSZ	Sun Feb 2019 11:46:32.393+0200
yyyy-MM-dd HH:mm:ss VV (JDK 8)	2019-02-24 11:45:41 Europe/Athens

До JDK 8 форматный шаблон можно применить посредством класса `SimpleDateFormat`:

```
// yyyy-MM-dd
Date date = new Date();
SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
String stringDate = formatter.format(date);
```

Начиная с JDK 8, форматный шаблон может быть применен посредством класса `DateTimeFormatter`:

◆ для `LocalDate` (дата без часового пояса в календарной системе ISO-8601):

```
// yyyy-MM-dd
LocalDate localDate = LocalDate.now();
DateTimeFormatter formatterLocalDate
    = DateTimeFormatter.ofPattern("yyyy-MM-dd");
String stringLD = formatterLocalDate.format(localDate);

// либо коротко
String stringLD = LocalDate.now()
    .format(DateTimeFormatter.ofPattern("yyyy-MM-dd"));
```

◆ для `LocalTime` (время без часового пояса в календарной системе ISO-8601):

```
// HH:mm:ss
LocalTime localTime = LocalTime.now();
DateTimeFormatter formatterLocalTime
    = DateTimeFormatter.ofPattern("HH:mm:ss");
String stringLT = formatterLocalTime.format(localTime);

// либо коротко
String stringLT = LocalTime.now()
    .format(DateTimeFormatter.ofPattern("HH:mm:ss"));
```

◆ для `LocalDateTime` (дата-время без часового пояса в календарной системе ISO-8601):

```
// yyyy-MM-dd HH:mm:ss
LocalDateTime localDateTime = LocalDateTime.now();
DateTimeFormatter formatterLocalDateTime
```

```
= DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
String stringLDT
    = formatterLocalDateTime.format(localDateTime);
```

```
// либо коротко
```

```
String stringLDT = LocalDateTime.now()
    .format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
```

◆ для ZonedDateTime (дата-время с часовым поясом в календарной системе ISO-8601):

```
// E MMM yyyy HH:mm:ss.SSSZ
ZonedDateTime zonedDateTime = ZonedDateTime.now();
DateTimeFormatter formatterZonedDateTime
    = DateTimeFormatter.ofPattern("E MMM yyyy HH:mm:ss.SSSZ");
String stringZDT
    = formatterZonedDateTime.format(zonedDateTime);
```

```
// либо коротко
```

```
String stringZDT = ZonedDateTime.now()
    .format(DateTimeFormatter
        .ofPattern("E MMM yyyy HH:mm:ss.SSSZ"));
```

◆ для OffsetDateTime (дата-время со смещением от UTC/GMT в календарной системе ISO-8601):

```
// E MMM yyyy HH:mm:ss.SSSZ
OffsetDateTime offsetDateTime = OffsetDateTime.now();
DateTimeFormatter formatterOffsetDateTime
    = DateTimeFormatter.ofPattern("E MMM yyyy HH:mm:ss.SSSZ");
String odt1 = formatterOffsetDateTime.format(offsetDateTime);
```

```
// либо коротко
```

```
String odt2 = OffsetDateTime.now()
    .format(DateTimeFormatter
        .ofPattern("E MMM yyyy HH:mm:ss.SSSZ"));
```

◆ для OffsetTime (время со смещением от UTC/GMT в календарной системе ISO-8601):

```
// HH:mm:ss,Z
OffsetTime offsetTime = OffsetTime.now();
DateTimeFormatter formatterOffsetTime
    = DateTimeFormatter.ofPattern("HH:mm:ss,Z");
String otl = formatterOffsetTime.format(offsetTime);
```

```
// либо коротко
```

```
String ot2 = OffsetTime.now()
    .format(DateTimeFormatter.ofPattern("HH:mm:ss,Z"));
```

60. Получение текущих даты и времени без времени или даты

До JDK 8 решение этой задачи было сосредоточено на классе `java.util.Date` (см. исходный код, прилагаемый к данной книге).

Начиная с JDK 8, дату и время можно получить посредством выделенных классов `LocalDate` и `LocalTime` из пакета `java.time`:

```
// 2019-02-24
LocalDate onlyDate = LocalDate.now();

// 12:53:28.812637300
LocalTime onlyTime = LocalTime.now();
```

61. Класс *LocalDateTime* из *LocalDate* и *LocalTime*

Класс `LocalDateTime` выставляет наружу серию методов `of()`, которые полезны для получения разного рода экземпляров класса `LocalDateTime`. Например, класс `LocalDateTime`, получаемый из года, месяца, дня, часа, минуты, секунды или наносекунды, выглядит следующим образом:

```
LocalDateTime ldt = LocalDateTime.of(2020, 4, 1, 12, 33, 21, 675);
```

Таким образом, приведенный выше фрагмент кода сочетает дату и время в качестве аргументов метода `of()`. Для того чтобы объединить дату и время как объекты, нужно воспользоваться следующей разновидностью метода `of()`:

```
public static LocalDateTime.of(LocalDate date, LocalTime time)
```

Это приводит к `LocalDate` и `LocalTime`, как показано ниже:

```
LocalDate localDate = LocalDate.now();    // 2019-Feb-24
LocalTime localTime = LocalTime.now();    // 02:08:10 PM
```

Их можно объединить в объект `LocalDateTime` следующим образом:

```
LocalDateTime localDateTime = LocalDateTime.of(localDate, localTime);
```

Форматирование `LocalDateTime` показывает дату и время:

```
// 2019-Feb-24 02:08:10 PM
String localDateTimeAsString = localDateTime
    .format(DateTimeFormatter.ofPattern("yyyy-MM-dd hh:mm:ss a"));
```

62. Машинное время посредством класса *Instant*

JDK 8 идет в комплекте с новым классом, который называется `java.time.Instant`. В сущности, класс `Instant` представляет собой мгновенную точку на временной шкале, начиная с первой секунды 1 января 1970 года (эпоха), в часовом поясе UTC с разрешающей способностью в наносекунду.



Класс `Instant` Java 8 по своей концепции аналогичен классу `java.util.Date`. Оба представляют момент на временной шкале в UTC. В то время как `Instant` имеет разрешающую способность до наносекунды, в `java.util.Date` она кратна миллисекунде.

Этот класс очень удобен для генерирования временных меток машинного времени. Для того чтобы получить такую метку времени, надо просто вызвать метод `now()`:

```
// 2019-02-24T15:05:21.781049600Z
```

```
Instant timestamp = Instant.now();
```

Аналогичный результат можно получить с помощью следующего фрагмента кода:

```
OffsetDateTime now = OffsetDateTime.now(ZoneOffset.UTC);
```

В качестве альтернативы можно использовать вот этот фрагмент кода:

```
Clock clock = Clock.systemUTC();
```



Метод `Instant.toString()` возвращает данные, которые соответствуют стандарту ISO-8601 для представления даты и времени.

Конвертирование значения *String* в объект *Instant*

Строка, которая соответствует стандарту ISO-8601 для представления даты и времени, может быть легко конвертирована в объект `Instant` посредством метода `Instant.parse()`, как в следующем ниже примере:

```
// 2019-02-24T14:31:33.197021300Z
```

```
Instant timestampFromString = Instant.parse("2019-02-24T14:31:33.197021300Z");
```

Прибавление времени к объекту *Instant* или вычитание времени из объекта *Instant*

Для добавления времени объект `Instant` имеет комплект методов. Например, прибавить 2 часа к текущей метке времени можно следующим образом:

```
Instant twoHourLater = Instant.now().plus(2, ChronoUnit.HOURS);
```

Для вычитания времени, например 10 минут, следует использовать вот такой фрагмент кода:

```
Instant tenMinutesEarlier = Instant.now()
    .minus(10, ChronoUnit.MINUTES);
```



Помимо метода `plus()`, класс `Instant` также содержит методы `plusNanos()`, `plusMillis()` и `plusSeconds()`. И кроме метода `minus()`, класс `Instant` также содержит методы `minusNanos()`, `minusMillis()` и `minusSeconds()`.

Сравнение объектов *Instant*

Сравнить два объекта *Instant* можно посредством методов *Instant.isAfter()* и *Instant.isBefore()*. Например, давайте рассмотрим следующие два объекта *Instant*:

```
Instant timestamp1 = Instant.now();
Instant timestamp2 = timestamp1.plusSeconds(10);
```

Проверить, что метка *timestamp1* показывает время после метки *timestamp2*, можно вот так:

```
boolean isAfter = timestamp1.isAfter(timestamp2); // false
```

Проверить, что метка *timestamp1* показывает время перед меткой *timestamp2*, можно вот так:

```
boolean isBefore = timestamp1.isBefore(timestamp2); // true
```

Разница во времени между двумя объектами *Instant* может быть вычислена посредством метода *Instant.until()*:

```
// 10 секунд
long difference = timestamp1.until(timestamp2, ChronoUnit.SECONDS);
```

Конвертирование из объекта *Instant* в *LocalDateTime*, *ZonedDateTime* и *OffsetDateTime* и обратно

Эти часто встречающиеся преобразования могут быть выполнены, как показано в следующих ниже примерах.

- ◆ Конвертирование между *Instant* и *LocalDateTime*. Поскольку *LocalDateTime* не имеет представления о часовом поясе, следует использовать нулевое смещение UTC+0:

```
// 2019-02-24T15:27:13.990103700
LocalDateTime ldt = LocalDateTime.ofInstant(Instant.now(), ZoneOffset.UTC);

// 2019-02-24T17:27:14.013105Z
Instant instantLDT = LocalDateTime.now().toInstant(ZoneOffset.UTC);
```

- ◆ Конвертирование между *Instant* и *ZonedDateTime*. Конвертирование *Instant* UTC+0 в *ZonedDateTime* UTC+1 часового пояса Парижа:

```
// 2019-02-24T16:34:36.138393100+01:00[Europe/Paris]
ZonedDateTime zdt = Instant.now().atZone(ZoneId.of("Europe/Paris"));

// 2019-02-24T16:34:36.150393800Z
Instant instantZDT = LocalDateTime.now()
    .atZone(ZoneId.of("Europe/Paris")).toInstant();
```

- ◆ Конвертирование между *Instant* и *OffsetDateTime*. Указать смещение на 2 часа:

```
// 2019-02-24T17:34:36.151393900+02:00
OffsetDateTime odt = Instant.now().atOffset(ZoneOffset.of("+02:00"));
```



```
// 2019-02-24T15:34:36.153394Z
Instant instantODT = LocalDateTime.now()
    .atOffset(ZoneOffset.of("+02:00")).toInstant();
```

63. Определение временного периода с использованием значений на основе даты и значений на основе времени

JDK 8 идет в комплекте с двумя новыми классами — `java.time.Period` и `java.time.Duration`. Давайте рассмотрим их подробно в следующих далее разделах.

Период времени с использованием значений на основе даты

Класс `Period` предназначен для представления количества времени с использованием значений на основе дат (лет, месяцев, недель и дней). Период времени можно получить разными способами. Например, период в 120 дней можно получить следующим образом:

```
Period fromDays = Period.ofDays(120); // P120D
```



Помимо метода `ofDays()`, класс `Period` также имеет методы `ofMonths()`, `ofWeeks()` и `ofYears()`.

Как вариант, к примеру, период в 2000 лет, 11 месяцев и 24 дня можно получить посредством метода `of()`, как показано ниже:

```
Period periodFromUnits = Period.of(2000, 11, 24); // P2000Y11M24D
```

Объект типа `Period` также можно получить из `LocalDate`:

```
LocalDate localDate = LocalDateTime.now();
Period periodFromLocalDate = Period.of(localDate.getYear(),
    localDate.getMonthValue(), localDate.getDayOfMonth());
```

Наконец, объект типа `Period` можно получить из объекта типа `String`, который совпадает с периодными форматами ISO-8601 `PnYnMnD` и `PnW`. Например, строка `P2019Y2M25D` представляет 2019 лет, 2 месяца и 25 дней:

```
Period periodFromString = Period.parse("P2019Y2M25D");
```



Вызов метода `Period.toString()` возвращает период, соблюдая при этом периодные форматы ISO-8601, `PnYnMnD` и `PnW` (например, `P120D`, `P2000Y11M24D`).

Но реальная сила класса `Period` проявляется, когда он используется для представления периода времени между двумя датами (например, датами `LocalDate`). Период

времени между 12 марта 2018 года и 20 июля 2019 года можно представить следующим образом:

```
LocalDate startLocalDate = LocalDate.of(2018, 3, 12);
LocalDate endLocalDate = LocalDate.of(2019, 7, 20);
Period periodBetween = Period.between(startLocalDate, endLocalDate);
```

Количество времени в годах, месяцах и днях можно получить посредством методов `Period.getYears()`, `Period.getMonths()` и `Period.getDays()` соответственно. Например, следующий ниже вспомогательный метод использует эти методы для вывода количества времени в виде строки:

```
public static String periodToYMD(Period period) {
    StringBuilder sb = new StringBuilder();
    sb.append(period.getYears())
      .append("y:")
      .append(period.getMonths())
      .append("m:")
      .append(period.getDays())
      .append("d");

    return sb.toString();
}
```

Давайте вызовем этот метод `periodToYMD` (разница составляет 1 год, 4 месяца и 8 дней):

```
periodToYMD(periodBetween); // 1y:4m:8d
```

Класс `Period` также полезен при выяснении того, является ли конкретная дата более ранней, чем другая дата. Для этого имеется флаговый метод с именем `isNegative()`. Имея период `A` и период `B`, можно получить отрицательный результат применения `Period.between(A, B)`, если `B` раньше `A`, или положительный, если `A` раньше `B`. Рассуждая логически в том же ключе, можно утверждать, что флаговый метод `isNegative()` возвращает `true`, если `B` раньше `A`, или `false`, если `A` раньше `B`, как в нашем случае, который вызывается в коде чуть позже (в сущности, этот метод возвращает `false`, если годы, месяцы или дни являются отрицательными):

```
// возвращает false, поскольку 12 марта 2018 раньше 20 июля 2019
periodBetween.isNegative();
```

Наконец, объект `Period` может быть модифицирован путем прибавления или вычитания периода времени. Для этого существуют такие методы, как `plusYears()`, `plusMonths()`, `plusDays()`, `minusYears()`, `minusMonths()` и `minusDays()`. Например, добавить 1 год к `periodBetween` можно следующим образом:

```
Period periodBetweenPlusYear = periodBetween.plusYears(1L);
```

Сложить два объекта `Period` можно посредством метода `Period.plus()`:

```
Period p1 = Period.ofDays(5);
Period p2 = Period.ofDays(20);
Period plp2 = p1.plus(p2); // P25D
```

Продолжительность времени с использованием значений на основе времени

Класс `Duration` предназначен для представления количества времени с использованием значений на основе времени (часов, минут, секунд или наносекунд). Эта продолжительность времени может быть получена разными способами. Например, продолжительность 10 часов может быть получена следующим образом:

```
Duration fromHours = Duration.ofHours(10); // PT10H
```



Помимо метода `ofHours()`, класс `Duration` также имеет методы `ofDays()`, `ofMillis()`, `ofMinutes()`, `ofSeconds()` и `ofNanos()`.

В качестве альтернативы, продолжительность 3 минуты может быть получена с помощью метода `of()` следующим образом:

```
Duration fromMinutes = Duration.of(3, ChronoUnit.MINUTES); // PT3M
```

Объект `Duration` также может быть получен из `LocalDateTime`:

```
LocalDateTime localDateTime = LocalDateTime.of(2018, 3, 12, 4, 14, 20, 670);  
  
// PT14M
```

```
Duration fromLocalDateTime = Duration.ofMinutes(localDateTime.getMinute());
```

Его также можно получить из `LocalTime`:

```
LocalTime localTime = LocalTime.of(4, 14, 20, 670);
```

```
// PT0.00000067S
```

```
Duration fromLocalTime = Duration.ofNanos(localTime.getNano());
```

Наконец, продолжительность можно получить из объекта `String`, который совпадает с форматом длительности ISO-8601 `PnDTnHnMn.nS`, где дни рассматриваются как состоящие ровно из 24 часов. Например, строка `P2DT3H4M` имеет 2 дня, 3 часа и 4 минуты:

```
Duration durationFromString = Duration.parse("P2DT3H4M");
```



Вызов метода `Duration.toString()` возвращает продолжительность, совпадающую с форматом длительности ISO-8601, `PnDTnHnMn.nS` (например, `PT10H`, `PT3M` или `PT51H4M`).

Но, как и в случае с классом `Period`, реальная сила класса `Duration` проявляется, когда он используется для представления периода между двумя моментами времени (например, объектами `Instant`). Продолжительность времени между 3 ноября 2015 г. 12:11:30 и 6 декабря 2016 г. 15:17:10 можно представить как разницу между двумя объектами `Instant` следующим образом:

```
Instant startInstant = Instant.parse("2015-11-03T12:11:30.00Z");
```

```
Instant endInstant = Instant.parse("2016-12-06T15:17:10.00Z");
```

```
// PT10059H5M40S
```

```
Duration durationBetweenInstant = Duration.between(startInstant, endInstant);
```

В секундах эту разницу можно получить посредством метода `Duration.getSeconds()`:

```
durationBetweenInstant.getSeconds(); // 36212740 секунд
```

Как вариант, продолжительность времени между 12 марта 2018 г., 04:14:20.000000670 и 20 июля 2019 г. 06:10:10.000000720 можно представить как разницу между двумя объектами `LocalDateTime` следующим образом:

```
LocalDateTime startLocalDateTime = LocalDateTime.of(2018, 3, 12, 4, 14, 20, 670);
LocalDateTime endLocalDateTime = LocalDateTime.of(2019, 7, 20, 6, 10, 10, 720);
```

```
// PT1881H55M50.00000005S или 42774950 секунд
```

```
Duration durationBetweenLDT = Duration.between(startLocalDateTime,
endLocalDateTime);
```

Наконец, продолжительность времени между 04:14:20.000000670 и 06:10:10.000000720 можно представить как разницу между двумя объектами `LocalTime` следующим образом:

```
LocalTime startLocalTime = LocalTime.of(4, 14, 20, 670);
LocalTime endLocalTime = LocalTime.of(6, 10, 10, 720);
```

```
// PT1H55M50.00000005S или 6950 секунд
```

```
Duration durationBetweenLT = Duration.between(startLocalTime, endLocalTime);
```

В предыдущих примерах продолжительность времени выражалась в секундах посредством метода `Duration.getSeconds()`, т. е. числом секунд в классе `Duration`. Однако класс `Duration` содержит набор методов, предназначенных для выражения продолжительность в других единицах времени — в днях посредством метода `toDays()`, в часах посредством метода `toHours()`, в минутах посредством метода `toMinutes()`, в миллисекундах посредством метода `toMillis()` и в наносекундах посредством метода `toNanos()`.

Конвертирование из одной единицы времени в другую может привести к образованию остатка. Так, конвертирование из секунд в минуты может привести к остатку секунд (например, 65 секунд — это 1 минута и 5 секунд, где 5 секунд — остаток). Остаток можно получить следующим набором методов: остаток в днях посредством метода `toDaysPart()`, остаток в часах посредством метода `toHoursPart()`, остаток в минутах посредством метода `toMinutesPart()` и т. д.

Допустим, что разницу нужно показывать в виде:

дни:часы:минуты:секунды:наносекунды

(например, 9d:2h:15m:20s:230n). Соединение усилий методов `toFoo()` и `toFooPart()` во вспомогательном методе приведет к следующему фрагменту кода:

```
public static String durationToDHMSN(Duration duration) {
    StringBuilder sb = new StringBuilder();
    sb.append(duration.toDays())
      .append("d:")
      .append(duration.toHoursPart())
      .append("h:")
```

```
.append(duration.toMinutesPart())
.append("m:")
.append(duration.toSecondsPart())
.append("s:")
.append(duration.toNanosPart())
.append("n");

return sb.toString();
}
```

Давайте вызовем этот метод `durationBetweenLDT` (разница составляет 495 дней, 1 час, 55 минут 50 секунд и 50 наносекунд):

```
// 495d:1h:55m:50s:50n
durationToDHMSN(durationBetweenLDT);
```

Аналогично классу `Period` класс `Duration` имеет флаговый метод с именем `isNegative()`. Этот метод полезен при выяснении того, является ли конкретное время более ранним, чем другое время. Имея продолжительность `A` и продолжительность `B`, можно получить отрицательный результат применения метода `Duration.between(A, B)`, если `B` раньше `A`, или положительный, если `A` раньше. Логически рассуждая дальше, можно сказать, что флаговый метод `isNegative()` возвращает `true`, если `B` раньше `A`, или `false`, если `A` раньше `B`, как в нашем случае:

```
durationBetweenLT.isNegative(); // false
```

Наконец, продолжительность можно модифицировать путем сложения или вычитания продолжительности времени. Для этого существуют такие методы, как `plusDays()`, `plusHours()`, `plusMinutes()`, `plusMillis()`, `plusNanos()`, `minusDays()`, `minusHours()`, `minusMinutes()`, `minusMillis()` и `minusNanos()`. Например, добавить 5 часов к `durationBetweenLT` можно следующим образом:

```
Duration durationBetweenPlus5Hours = durationBetweenLT.plusHours(5);
```

Сложить два класса `Duration` можно посредством метода `Duration.plus()`:

```
Duration d1 = Duration.ofMinutes(20);
Duration d2 = Duration.ofHours(2);
```

```
Duration dld2 = d1.plus(d2);
```

```
System.out.println(d1 + "+" + d2 + "=" + dld2); // PT2H20M
```

64. Получение единиц даты и времени

Для объекта `Date` решение может опираться на экземпляр класса `Calendar`. Это решение содержится в исходном коде, прилагаемом к данной книге.

Для классов JDK 8 язык Java предусматривает выделенные методы `getFoo()` и метод `get(TemporalField field)`. Например, допустим, что у нас есть следующий объект `LocalDateTime`:

```
LocalDateTime ldt = LocalDateTime.now();
```

Опираясь на методы `getFoo()`, мы получаем следующий ниже фрагмент кода:

```
int year = ldt.getYear();
int month = ldt.getMonthValue();
int day = ldt.getDayOfMonth();
int hour = ldt.getHour();
int minute = ldt.getMinute();
int second = ldt.getSecond();
int nano = ldt.getNano();
```

Как вариант, опираясь на метод `get(TemporalField поле)`, мы получим следующее:

```
int yearLDT = ldt.get(ChronoField.YEAR);
int monthLDT = ldt.get(ChronoField.MONTH_OF_YEAR);
int dayLDT = ldt.get(ChronoField.DAY_OF_MONTH);
int hourLDT = ldt.get(ChronoField.HOUR_OF_DAY);
int minuteLDT = ldt.get(ChronoField.MINUTE_OF_HOUR);
int secondLDT = ldt.get(ChronoField.SECOND_OF_MINUTE);
int nanoLDT = ldt.get(ChronoField.NANO_OF_SECOND);
```

Обратите внимание, что месяцы отсчитываются от единицы, т. е. от января.

Например, объект `LocalDateTime` со значением `2019-02-25T12:58:13.109389100` можно разрезать на единицы даты и времени, в результате получив следующее:

```
Year: 2019 Month: 2 Day: 25 Hour: 12 Minute: 58 Second: 13 Nano:
109389100
```

Имея немного интуиции и документацию, очень легко адаптировать этот пример для `LocalDate`, `LocalTime`, `ZonedDateTime` и других объектов.

65. Прибавление к дате и времени и вычитание из них

Решение этой задачи опирается на API-интерфейсы Java, которые предназначены для управления датой и временем. Давайте рассмотрим их в следующих далее разделах.

Работа с датой

Для объекта `Date` решение может опираться на экземпляр класса `Calendar`. Это решение содержится в исходном коде, прилагаемом к данной книге.

Работа с `LocalDateTime`

С переходом к JDK 8 основное внимание уделяется классам `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant` и многим другим. Новый API даты-времени Java идет в комплекте с методами, которые предназначены для сложения или вычитания количества времени. `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `OffsetDateTime`, `Instant`, `Period`, `Duration` и многие другие идут в комплекте с такими методами, как `plusFoo()`

и `minusFoo()`, где `Foo` можно заменить единицей времени (например, `plusYears()`, `plusMinutes()`, `minusHours()`, `minusSeconds()` и т. д.).

Допустим, что у нас есть следующий объект `LocalDateTime`:

```
// 2019-02-25T14:55:06.651155500
LocalDateTime ldt = LocalDateTime.now();
```

Добавить 10 минут так же просто, как вызвать `LocalDateTime.plusMinutes(long minutes)`, а вычесть 10 минут так же просто, как вызвать `LocalDateTime.minusMinutes(long minutes)`:

```
LocalDateTime ldtAfterAddingMinutes = ldt.plusMinutes(10);
LocalDateTime ldtAfterSubtractingMinutes = ldt.minusMinutes(10);
```

Результат вывевит следующие ниже даты:

```
После прибавления 10 минут: 2019-02-25T15:05:06.651155500
После вычитания 10 минут: 2019-02-25T14:45:06.651155500
```



Помимо методов, выделенных для каждой единицы времени, эти классы также поддерживают `plus/minus(TemporalAmount amountToAdd)` и `plus/minus(long amountToAdd, TemporalUnit unit)`.

Теперь давайте сосредоточимся на классе `Instant`. Помимо методов `plus/minusSeconds()`, `plus/minusMillis()` и `plus/minusNanos()`, класс `Instant` также предусматривает метод `plus/minus(TemporalAmount amountToAdd)`.

Для того чтобы проиллюстрировать этот метод, давайте допустим, что у нас есть следующий ниже объект `Instant`:

```
// 2019-02-25T12:55:06.654155700Z
Instant timestamp = Instant.now();
```

Теперь давайте добавим и вычтем 5 часов:

```
Instant timestampAfterAddingHours = timestamp.plus(5, ChronoUnit.HOURS);
Instant timestampAfterSubtractingHours = timestamp.minus(5, ChronoUnit.HOURS);
```

Результат покажет следующие значения объекта `Instant`:

```
После добавления 5 часов: 2019-02-25T17:55:06.654155700Z
После вычитания 5 часов: 2019-02-25T07:55:06.654155700Z
```

66. Получение всех часовых поясов в UTC и GMT

UTC (всемирное координированное время) и GMT (среднее время по Гринвичу) признаны стандартными точками отсчета для работы с датами и временем. Сегодня UTC предпочтительнее, но UTC и GMT должны возвращать один и тот же результат в большинстве случаев.

Для того чтобы получить все часовые пояса в UTC и GMT, решение задачи должно сосредоточиться на имплементации до и после JDK 8. Итак, давайте начнем с решения, которое широко использовалось до JDK 8.

До JDK 8

Решение задачи должно извлекать доступные идентификаторы часовых поясов (Африка/Бамако, Европа/Белград и т. д.). Кроме того, каждый идентификатор часового пояса должен использоваться для создания объекта `TimeZone`. Наконец, решение должно извлекать смещение, которое было специфичным для каждого часового пояса, принимая во внимание переход на летнее/зимнее время. Это решение содержится в исходном коде, прилагаемом к данной книге.

Начиная с JDK 8

Новый API даты-времени Java предусматривает новые рычаги для решения этой задачи.

На первом шаге доступные идентификаторы часовых поясов можно получить посредством класса `ZoneId` следующим образом:

```
Set<String> zoneIds = ZoneId.getAvailableZoneIds();
```

На втором шаге каждый идентификатор часового пояса должен использоваться для создания экземпляра класса `ZoneId`. Это может быть достигнуто посредством метода `ZoneId.of(String zoneId)`:

```
ZoneId zoneid = ZoneId.of(current_zone_id);
```

На третьем шаге каждый объект `ZoneId` можно использовать для получения времени, специфичного для определенного часового пояса. Это означает, что в качестве "лабораторной мыши" необходима опорная дата-время. Эта опорная дата-время (без часового пояса, `LocalDateTime.now()`) комбинируются с заданным часовым поясом (`ZoneId`), посредством метода `LocalDateTime.atZone()`, для того чтобы получить объект `ZonedDateTime` (дата-время, которая знает о часовом поясе):

```
LocalDateTime now = LocalDateTime.now();
ZonedDateTime zdt = now.atZone(ZoneId.of(zone_id_instance));
```



Метод `atZone()` максимально точно сочетается с датой и временем, принимая во внимание правила часовых поясов, например летнее/зимнее время.

На четвертом шаге исходный код может использовать объект `ZonedDateTime` для извлечения смещения от UTC (например, для Европы/Бухареста смещение от UTC равно `+02:00`):

```
String utcOffset = zdt.getOffset().getId().replace("Z", "+00:00");
```

Метод `getId()` возвращает идентификатор смещения нормализованного часового пояса. Смещение `+00:00` возвращается как символ `Z`; поэтому код должен быстро заменить `Z` на `+00:00`, чтобы выровняться с остальными смещениями, которые совпадают с форматом `+hh:mm` или `+hh:mm:ss`.

Теперь давайте соединим эти шаги вместе во вспомогательном методе:

```
public static List<String> fetchTimeZones(OffsetType type) {
    List<String> timezones = new ArrayList<>();
    Set<String> zoneIds = ZoneId.getAvailableZoneIds();
    LocalDateTime now = LocalDateTime.now();

    zoneIds.forEach((zoneId) -> {
        timezones.add("(" + type + now.atZone(ZoneId.of(zoneId))
            .getOffset().getId().replace("Z", "+00:00") + ") " + zoneId);
    });

    return timezones;
}
```

Допустив, что этот метод располагается в классе `DateTimes`, получаем следующий фрагмент кода:

```
List<String> timezones
    = DateTimes.fetchTimeZones(DateTimes.OffsetType.GMT);
Collections.sort(timezones); // необязательная сортировка
timezones.forEach(System.out::println);
```

В дополнение к этому выходной снимок отображается следующим образом:

```
(GMT+00:00) Africa/Abidjan
(GMT+00:00) Africa/Accra
(GMT+00:00) Africa/Bamako
...
(GMT+11:00) Australia/Tasmania
(GMT+11:00) Australia/Victoria
...
```

67. Получение локальных даты и времени во всех имеющихся часовых поясах

Решение этой задачи можно получить, выполнив следующие шаги:

1. Получить местные время и дату.
2. Получить имеющиеся часовые пояса.
3. До JDK 8 использовать класс `SimpleDateFormat` с методом `setTimeZone()`.
4. Начиная с JDK 8, использовать класс `ZonedDateTime`.

До JDK 8

До JDK 8 быстрое решение задачи получения текущих локальных даты и времени — вызвать пустой конструктор класса `Date`. Далее использовать объект `Date` для показа его во всех имеющихся часовых поясах, которые можно получить посредством класса `TimeZone`. Это решение содержится в исходном коде, прилагаемом к данной книге.

Начиная с JDK 8

Начиная с JDK 8, удобным решением задачи получения текущих локальных даты и времени в часовом поясе по умолчанию является вызов метода `ZonedDateTime.now()`:

```
ZonedDateTime zlt = ZonedDateTime.now();
```

Переменная `zlt` — это текущая дата в часовом поясе по умолчанию. Далее эта дата должна показываться во всех имеющихся часовых поясах, получаемых посредством класса `ZoneId`:

```
Set<String> zoneIds = ZoneId.getAvailableZoneIds();
```

Наконец, исходный код может перебрать в цикле идентификаторы `zoneId` и для каждого идентификатора часового пояса вызвать метод `ZonedDateTime.withZoneSameInstant(ZoneId zone)`. Указанный метод возвращает копию этой даты и времени с другим часовым поясом, сохраняя объект `Instant`:

```
public static List<String> localTimeToAllTimeZones() {
    List<String> result = new ArrayList<>();
    Set<String> zoneIds = ZoneId.getAvailableZoneIds();
    DateTimeFormatter formatter
        = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm:ss a Z");
    ZonedDateTime zlt = ZonedDateTime.now();

    zoneIds.forEach((zoneId) -> {
        result.add(zlt.format(formatter) + " in " + zoneId + " is "
            + zlt.withZoneSameInstant(ZoneId.of(zoneId))
                .format(formatter));
    });

    return result;
}
```

Выходной снимок этого метода может быть следующим:

```
2019-Feb-26T14:26:30 PM +0200 in Africa/Nairobi
is 2019-Feb-26T15:26:30 PM +0300
2019-Feb-26T14:26:30 PM +0200 in America/Marigot
is 2019-Feb-26T08:26:30 AM -0400
...
2019-Feb-26T14:26:30 PM +0200 in Pacific/Samoa
is 2019-Feb-26T01:26:30 AM -1100
```

68. Вывод на экран информации о дате и времени полета

Решение, представленное в данном разделе, покажет следующую информацию о полете из Перта (Австралия) в Бухарест (Европа) продолжительностью 15 часов 30 минут:

- ♦ дата-время вылета и прилета UTC;
- ♦ дата-время вылета и прилета в Бухарест по местному времени Перта;
- ♦ дата-время вылета и прилета в Бухарест по местному времени Бухареста.

Допустим, что ориентировочными датой и временем вылета из Перта является 26 февраля 2019 года 16:00 (или 4:00 вечера):

```
LocalDateTime ldt = LocalDateTime.of(2019, Month.FEBRUARY, 26, 16, 00);
```

Сначала давайте совместим эти дату и время с часовым поясом Австралии/Перта (+08:00). В результате получим объект `ZonedDateTime`, который является специфичным для Австралии/Перта (это будет датой и временем вылета по местному времени Перта):

```
// 04:00 PM, Feb 26, 2019 +0800 Australia/Perth
```

```
ZonedDateTime auPerthDepart = ldt.atZone(ZoneId.of("Australia/Perth"));
```

Далее прибавим 15 часов 30 минут к объекту `ZonedDateTime`. Полученный объект `ZonedDateTime` представляет собой дату и время по местному времени Перта (это будет датой и временем прилета в Бухарест по местному времени Перта):

```
// 07:30 AM, Feb 27, 2019 +0800 Australia/Perth
```

```
ZonedDateTime auPerthArrive = auPerthDepart.plusHours(15).plusMinutes(30);
```

Теперь рассчитаем дату и время вылета из Перта по местному времени Бухареста. В сущности, следующий ниже фрагмент кода выражает дату и время вылета из часового пояса Перта в часовой пояс Бухареста:

```
// 10:00 AM, Feb 26, 2019 +0200 Europe/Bucharest
```

```
ZonedDateTime euBucharestDepart
```

```
= auPerthDepart.withZoneSameInstant(ZoneId.of("Europe/Bucharest"));
```

Наконец, рассчитаем дату и время прилета по местному времени Бухареста. Следующий ниже фрагмент кода выражает дату и время прилета из часового пояса Перта в часовой пояс Бухареста:

```
// 01:30 AM, Feb 27, 2019 +0200 Europe/Bucharest
```

```
ZonedDateTime euBucharestArrive
```

```
= auPerthArrive.withZoneSameInstant(ZoneId.of("Europe/Bucharest"));
```

Как показано на рис. 3.1, UTC-время вылета из Перта равно 8:00 (утро), а UTC-время прилета в Бухарест равно 11:30 вечера.

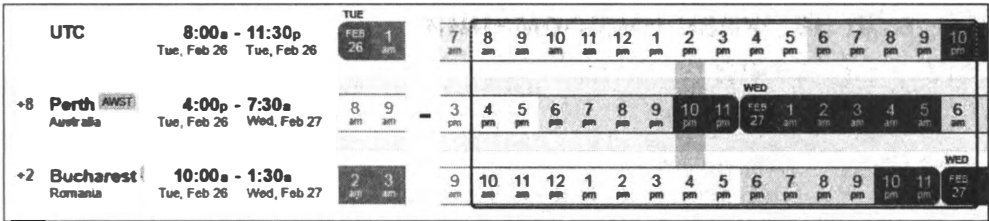


Рис. 3.1

Эти времена можно легко извлечь как `OffsetDateTime` следующим образом:

```
// 08:00 AM, Feb 26, 2019
OffsetDateTime utcAtDepart = auPerthDepart.withZoneSameInstant(
    ZoneId.of("UTC")).toOffsetDateTime();

// 11:30 PM, Feb 26, 2019
OffsetDateTime utcAtArrive = auPerthArrive.withZoneSameInstant(
    ZoneId.of("UTC")).toOffsetDateTime();
```

69. Конвертирование временной метки UNIX в дату и время

В целях решения данной задачи давайте предположим, что имеется следующая временная метка UNIX — 1573768800. Она эквивалентна информации, представленной ниже:

- ◆ 11/14/2019 @ 10:00pm (UTC);
- ◆ 2019-11-14T22:00:00+00:00 in ISO-8601;
- ◆ Thu, 14 Nov 2019 22:00:00 +0000 in RFC 822, 1036, 1123, 2822;
- ◆ Thursday, 14-Nov-19 22:00:00 UTC in RFC 2822;
- ◆ 2019-11-14T22:00:00+00:00 in RFC 3339.

Для того чтобы конвертировать временную метку UNIX в дату и время, важно знать, что разрешающая способность временных меток UNIX равна секундам, тогда как класс `java.util.Date` требует миллисекунд. Поэтому решение задачи получения объекта `Date` из временной метки UNIX требует конверсии из секунд в миллисекунды путем умножения временной метки UNIX на 1000, как показано в следующих ниже двух примерах:

```
long unixTimestamp = 1573768800;

// Fri Nov 15 00:00:00 EET 2019 - в часовом поясе по умолчанию
Date date = new Date(unixTimestamp * 1000L);

// Fri Nov 15 00:00:00 EET 2019 - в часовом поясе по умолчанию
Date date = new Date(TimeUnit.MILLISECONDS
    .convert(unixTimestamp, TimeUnit.SECONDS));
```

Начиная с JDK 8, класс `Date` использует метод `from(Instant instant)`. Кроме того, класс `Instant` идет в комплекте с методом `ofEpochSecond(long epochSecond)`, который возвращает экземпляр класса `Instant`, используя заданные секунды из эпохи, 1970-01-01T00:00:00Z:

```
// 2019-11-14T22:00:00Z in UTC
Instant instant = Instant.ofEpochSecond(unixTimestamp);

// Fri Nov 15 00:00:00 EET 2019 - в часовом поясе по умолчанию
Date date = Date.from(instant);
```

Объект `Instant`, полученный в приведенном выше примере, можно использовать для создания объектов `LocalDateTime` или `ZonedDateTime`:

```
// 2019-11-15T06:00
LocalDateTime date = LocalDateTime
    .ofInstant(instant, ZoneId.of("Australia/Perth"));

// 2019-Nov-15 00:00:00 +0200 Europe/Bucharest
ZonedDateTime date = ZonedDateTime
    .ofInstant(instant, ZoneId.of("Europe/Bucharest"));
```

70. Отыскание первого/последнего дня месяца

Правильное решение этой задачи будет опираться на интерфейсы `Temporal` и `TemporalAdjuster` JDK 8.

Интерфейс `Temporal` лежит в основе представлений даты/времени. Другими словами, классы, представляющие дату и/или время, имплементируют этот интерфейс. Например, ниже перечислено всего несколько классов, имплементирующих этот интерфейс:

- ◆ `LocalDate` (дата без часового пояса в календарной системе ISO-8601);
- ◆ `LocalTime` (время без часового пояса в календарной системе ISO-8601);
- ◆ `LocalDateTime` (дата и время без часового пояса в календарной системе ISO-8601);
- ◆ `ZonedDateTime` (дата и время с часовым поясом в календарной системе ISO-8601);
- ◆ `OffsetDateTime` (дата и время со смещением от UTC/GMT в календарной системе ISO-8601);
- ◆ `HijrahDate` (дата в календарной системе хиджры).

Класс `TemporalAdjuster` — это функциональный интерфейс, определяющий стратегии, которые можно использовать для настройки объекта `Temporal`. Помимо возможности определения индивидуально настроенных стратегий, класс `TemporalAdjuster` предоставляет несколько предопределенных стратегий, как показано ниже (документация содержит весь список, который впечатляет):

- ◆ `firstDayOfMonth()` (возвращает первый день текущего месяца);
- ◆ `lastDayOfMonth()` (возвращает последний день текущего месяца);

- ◆ `firstDayOfNextMonth()` (возвращает первый день следующего месяца);
- ◆ `firstDayOfNextYear()` (возвращает первый день следующего года).

Обратите внимание, что первые два настройщика в приведенном выше списке являются как раз теми, которые необходимы для решения этой задачи.

Рассмотрим вариант решения с `LocalDate`:

```
LocalDate date = LocalDate.of(2019, Month.FEBRUARY, 27);
```

И давайте возьмем первый/последний дни февраля:

```
// 2019-02-01
```

```
LocalDate firstDayOfFeb = date.with(TemporalAdjusters.firstDayOfMonth());
```

```
// 2019-02-28
```

```
LocalDate lastDayOfFeb = date.with(TemporalAdjusters.lastDayOfMonth());
```

Похоже, что использовать предопределенные стратегии довольно просто. Но давайте допустим, что в задаче требуется найти дату, которая находится через 21 день после 27 февраля 2019 года, т. е. 20 марта 2019 года. Для этой задачи не существует предопределенной стратегии, поэтому необходима специализированная стратегия. Решение может основываться на лямбда-выражении, как в следующем вспомогательном ниже методе:

```
public static LocalDate getDayAfterDays(
    LocalDate startDate, int days) {

    Period period = Period.ofDays(days);
    TemporalAdjuster ta = p -> p.plus(period);
    LocalDate endDate = startDate.with(ta);

    return endDate;
}
```

Если этот метод располагается в классе `DateTimes`, то следующий ниже вызов вернет ожидаемый результат:

```
// 2019-03-20
```

```
LocalDate datePlus21Days = DateTimes.getDayAfterDays(date, 21);
```

Следуя тому же техническому приему, но опираясь на статический фабричный метод `ofDateAdjuster()`, приведенный далее фрагмент кода определяет статического настройщика, который возвращает следующую дату, приходящуюся на субботу:

```
static TemporalAdjuster NEXT_SATURDAY
    = TemporalAdjusters.ofDateAdjuster(today -> {

    DayOfWeek dayOfWeek = today.getDayOfWeek();

    if (dayOfWeek == DayOfWeek.SATURDAY) {
        return today;
    }
```

```
if (dayOfWeek == DayOfWeek.SUNDAY) {
    return today.plusDays(6);
}

return today.plusDays(6 - dayOfWeek.getValue());
});
```

Давайте вызовем этот метод для 27 февраля 2019 года (следующая суббота приходится на 2 марта 2019 года):

```
// 2019-03-02
LocalDate nextSaturday = date.with(NEXT_SATURDAY);
```

Наконец, этот функциональный интерфейс определяет абстрактный метод `adjustInto()`, который может быть переопределен в специализированных имплементациях путем передачи ему объекта `Temporal` следующим образом:

```
public class NextSaturdayAdjuster implements TemporalAdjuster {
    @Override
    public Temporal adjustInto(Temporal temporal) {

        DayOfWeek dayOfWeek = DayOfWeek
            .of(temporal.get(ChronoField.DAY_OF_WEEK));

        if (dayOfWeek == DayOfWeek.SATURDAY) {
            return temporal;
        }

        if (dayOfWeek == DayOfWeek.SUNDAY) {
            return temporal.plus(6, ChronoUnit.DAYS);
        }

        return temporal.plus(6 - dayOfWeek.getValue(), ChronoUnit.DAYS);
    }
}
```

Вот пример его использования:

```
NextSaturdayAdjuster nsa = new NextSaturdayAdjuster();

// 2019-03-02
LocalDate nextSaturday = date.with(nsa);
```

71. Определение/извлечение поясных смещений

Под поясным смещением мы понимаем количество времени, которое необходимо прибавить ко времени GMT/UTC или вычесть из него, чтобы получить дату и время для конкретного часового пояса на земном шаре (например, Перт в Австралии). Обычно поясное смещение печатается в виде фиксированного числа часов и минут: +02:00, -08:30, +0400, UTC+01:00 и т. д.

Одним словом, поясное смещение — это время, на которое часовой пояс отличается от GMT/UTC.

До JDK 8

До JDK 8 часовой пояс можно было определить посредством класса `java.util.TimeZone`. С этим классом исходный код может получить поясное смещение посредством метода `TimeZone.getRawOffset()` (сырая — Raw — часть имени происходит из того факта, что этот метод не учитывает переход на летнее/зимнее время). Это решение содержится в исходном коде, прилагаемом к данной книге.

Начиная с JDK 8

Начиная с JDK 8, за работу с представлениями часовых поясов отвечают два класса. Во-первых, это класс `java.time.ZoneId`, который представляет часовой пояс, такой как Афины (Европа), и во-вторых, это класс `java.time.ZoneOffset` (расширяющий класс `ZoneId`), который представляет фиксированное количество времени (смещение) конкретного часового пояса от GMT/UTC.

Новый API даты и времени Java по умолчанию регулирует переход на летнее/зимнее время; поэтому регион с летне-зимними циклами, использующий летнее/зимнее время, будет иметь два класса `ZoneOffset`.

Поясное смещение от UTC можно легко получить следующим образом (это +00:00, представленное в Java символом Z):

```
// Z
ZoneOffset zoneOffsetUTC = ZoneOffset.UTC;
```

Системный часовой пояс, заданный по умолчанию, также можно получить с помощью класса `ZoneOffset`:

```
// Europe/Athens
ZoneId defaultZoneId = ZoneOffset.systemDefault();
```

Для того чтобы учесть поясное смещение с переходом на летнее/зимнее время, исходный код должен связать с ним дату и время. Например, свяжем класс `LocalDateTime` (можно также использовать `Instant`) следующим образом:

```
// он регулирует переход на летнее/зимнее время по умолчанию
LocalDateTime ldt = LocalDateTime.of(2019, 6, 15, 0, 0);
ZoneId zoneId = ZoneId.of("Europe/Bucharest");
```

```
// +03:00
ZoneOffset zoneOffset = zoneId.getRules().getOffset(ldt);
```

Поясное смещение также можно задать строкой. Например, следующий фрагмент кода получает поясное смещение +02:00:

```
ZoneOffset zoneOffsetFromString = ZoneOffset.of("+02:00");
```


Этот подход очень удобен для быстрого прибавления поясного смещения к объекту `Temporal`, поддерживающему поясное смещение. Например, его следует использовать для прибавления поясного смещения в `OffsetTime` и `OffsetDateTime` (удобные способы хранения даты в базе данных или отправки по сетям):

```
OffsetTime offsetTime = OffsetTime.now(zoneOffsetFromString);
OffsetDateTime offsetDateTime = OffsetDateTime.now(zoneOffsetFromString);
```

Еще одно решение нашей задачи состоит в том, чтобы опереться на определение объекта `ZoneOffset` из часов, минут и секунд. Этому посвящен один из вспомогательных методов `ZoneOffset`:

```
// +08:30 (это было получено из 8 часов и 30 минут)
ZoneOffset zoneOffsetFromHoursMinutes = ZoneOffset.ofHoursMinutes(8, 30);
```



Помимо метода `ZoneOffset.ofHoursMinutes()`, имеются методы `ZoneOffset.ofHours()`, `ofHoursMinutesSeconds()` и `ofTotalSeconds()`.

Наконец, каждый объект `Temporal`, поддерживающий поясное смещение, предусматривает удобный метод `getOffset()`. Например, следующий ниже фрагмент кода получает поясное смещение из указанного выше объекта `offsetDateTime`:

```
// +02:00
ZoneOffset zoneOffsetFromOdt = offsetDateTime.getOffset();
```

72. Конвертирование из даты во время и наоборот

Представленное здесь решение будет охватывать следующие классы, имплементирующие интерфейс `Temporal`: `Instant`, `LocalDate`, `LocalDateTime`, `ZonedDateTime`, `OffsetDateTime`, `LocalTime` и `OffsetTime`.

Date — Instant

В конвертировании из `Date` в `Instant` решение может опираться на метод `Date.toInstant()`. Обратное можно осуществить посредством метода `Date.from(Instant instant)`.

◆ Конвертировать из `Date` в `Instant` можно следующим образом:

```
Date date = new Date();

// например, 2019-02-27T12:02:49.369Z, UTC
Instant instantFromDate = date.toInstant();
```

◆ Конвертировать из `Instant` в `Date` можно следующим образом:

```
Instant instant = Instant.now();

// Wed Feb 27 14:02:49 EET 2019, системный часовой пояс по умолчанию
Date dateFromInstant = Date.from(instant);
```



Имейте в виду, что `Date` не знает о часовом поясе, но показывается в системном часовом поясе, установленном по умолчанию (например, посредством метода `toString()`). `Instant` показывается с часовым поясом UTC.

Давайте быстро обернем эти фрагменты кода в два служебных метода, определенных в служебном классе — `DateConverters`:

```
public static Instant dateToInstant(Date date) {
    return date.toInstant();
}

public static Date instantToDate(Instant instant) {
    return Date.from(instant);
}
```

Далее, дополним этот класс методами, перечисленными на рис. 3.2.

DEFAULT_TIME_ZONE	ZoneId
dateToInstant(Date date)	Instant
dateToLocalDate(Date date)	LocalDate
dateToLocalDateTime(Date date)	LocalDateTime
dateToLocalTime(Date date)	LocalTime
dateToOffsetDateTime(Date date)	OffsetDateTime
dateToOffsetTime(Date date)	OffsetTime
dateToZonedDateTime(Date date)	ZonedDateTime
instantToDate(Instant instant)	Date
localDateTimeToDate(LocalDateTime localDateTime)	Date
localDateToDate(LocalDate localDate)	Date
localTimeToDate(LocalTime localTime)	Date
offsetDateTimeToDate(OffsetDateTime offsetDateTime)	Date
offsetTimeToDate(OffsetTime offsetTime)	Date
zonedDateTimeToDate(ZonedDateTime zonedDateTime)	Date

Рис. 3.2

Константа из снимка экрана `DEFAULT_TIME_ZONE` является системным часовым поясом по умолчанию:

```
public static final ZoneId DEFAULT_TIME_ZONE = ZoneId.systemDefault();
```

Date — LocalDate

`Date` можно конвертировать в `LocalDate` посредством объекта `Instant`. После того как мы получили объект `Instant` из заданного объекта `Date`, решение может применить к нему системный часовой пояс по умолчанию и вызвать метод `toLocalDate()`:

```
// например, 2019-03-01
public static LocalDate dateToLocalDate(Date date) {
    return dateToInstant(date).atZone(DEFAULT_TIME_ZONE).toLocalDate();
}
```

Конвертирование из `LocalDate` в `Date` должно учитывать, что `LocalDate` не содержит компонент времени в качестве `Date`, поэтому такое решение должно предоставить компонент времени в качестве начала дня (более подробную информацию об этом можно найти в разд. 75 "Начало и конец дня" далее в этой главе):

```
// например, Fri Mar 01 00:00:00 EET 2019
public static Date localDateToDate(LocalDate localDate) {
    return Date.from(localDate.atStartOfDay(
        DEFAULT_TIME_ZONE).toInstant());
}
```

Date — DateLocalTime

Конвертирование из `Date` в `DateLocalTime` похоже на конвертирование из `Date` в `LocalDate`, за исключением того, что это решение должно вызывать метод `toLocalDateTime()` следующим образом:

```
// например, 2019-03-01T07:25:25.624
public static LocalDateTime dateToLocalDateTime(Date date) {
    return dateToInstant(date).atZone(
        DEFAULT_TIME_ZONE).toLocalDateTime();
}
```

Конвертирование из `LocalDateTime` в `Date` является прямолинейным. Нужно просто применить системный часовой пояс по умолчанию и вызвать метод `toInstant()`:

```
// например, Fri Mar 01 07:25:25 EET 2019
public static Date localDateTimeToDate(LocalDateTime localDateTime) {
    return Date.from(localDateTime.atZone(
        DEFAULT_TIME_ZONE).toInstant());
}
```

Date — ZonedDateTime

Конвертировать из `Date` в `ZonedDateTime` можно посредством объекта `Instant`, полученного из заданного объекта `Date`, и системного часового пояса по умолчанию:

```
// например, 2019-03-01T07:25:25.624+02:00[Europe/Athens]
public static ZonedDateTime dateToZonedDateTime(Date date) {
    return dateToInstant(date).atZone(DEFAULT_TIME_ZONE);
}
```

Конвертирование из `ZonedDateTime` в `Date` связано с конвертированием `ZonedDateTime` в `Instant`:

```
// например, Fri Mar 01 07:25:25 EET 2019
public static Date zonedDateTimeToDate(ZonedDateTime zonedDateTime) {
    return Date.from(zonedDateTime.toInstant());
}
```

Date — OffsetDateTime

Конвертирование из Date в OffsetDateTime опирается на метод `toOffsetDateTime()`:

```
// например, 2019-03-01T07:25:25.624+02:00
public static OffsetDateTime dateToOffsetDateTime(Date date) {
    return dateToInstant(date).atZone(
        DEFAULT_TIME_ZONE).toOffsetDateTime();
}
```

Для конвертирования из OffsetDateTime в Date требуется выполнить два шага. Во-первых, конвертировать OffsetDateTime в LocalDateTime. Во-вторых, конвертировать LocalDateTime в Instant с соответствующим смещением:

```
// например, Fri Mar 01 07:55:49 EET 2019
public static Date offsetDateTimeToDate(OffsetDateTime offsetDateTime) {

    return Date.from(offsetDateTime.toLocalDateTime()
        .toInstant(ZoneOffset.of(offsetDateTime.getOffset().getId())));
}
```

Date — LocalTime

Конвертирование из Date в LocalTime опирается на метод `LocalTime.toInstant()` следующим образом:

```
// например, 08:03:20.336
public static LocalTime dateToLocalTime(Date date) {
    return LocalTime.ofInstant(dateToInstant(date), DEFAULT_TIME_ZONE);
}
```

Конвертирование LocalTime в Date должно учитывать, что LocalTime не имеет компонента даты. Это означает, что решением должна быть установлена дата 1 января 1970 года в качестве эпохи:

```
// например, Thu Jan 01 08:03:20 EET 1970
public static Date localTimeToDate(LocalTime localTime) {
    return Date.from(localTime.atDate(LocalDate.EPOCH)
        .toInstant(DEFAULT_TIME_ZONE.getRules()
        .getOffset(Instant.now())));
}
```

Date — OffsetTime

Конвертирование из Date в OffsetTime опирается на метод `OffsetTime.toInstant()` следующим образом:

```
// например, 08:03:20.336+02:00
public static OffsetTime dateToOffsetTime(Date date) {
    return OffsetTime.ofInstant(dateToInstant(date), DEFAULT_TIME_ZONE);
}
```

Конвертирование из `OffsetTime` в `Date` должно учитывать, что `OffsetTime` не имеет компонента даты. Это означает, что решение должно установить дату на 1 января 1970 года в качестве эпохи:

```
// например, Thu Jan 01 08:03:20 EET 1970
public static Date offsetTimeToDate(OffsetTime offsetTime) {
    return Date.from(offsetTime.atDate(LocalDate.EPOCH).toInstant());
}
```

73. Обход интервала дат в цикле

Допустим, что интервал ограничен датой начала 2019 Feb 1 и датой окончания 2019 Feb 21. Решение этой задачи должно использовать цикл с обходом интервала [2019 Feb 1; 2019 Feb 21) с шагом в один день и печатать каждую дату на экране. В принципе, нужно решить главные подзадачи:

- ♦ прекратить цикл, как только дата начала будет равна дате окончания;
- ♦ увеличивать начальную дату на один день, пока не будет достигнута конечная дата.

До JDK 8

До JDK 8 решение опиралось на служебный класс `Calendar`. Это решение содержится в исходном коде, прилагаемом к данной книге.

Начиная с JDK 8

Начиная с JDK 8, даты можно легко определять как `LocalDate` без помощи класса `Calendar`:

```
LocalDate startLocalDate = LocalDate.of(2019, 2, 1);
LocalDate endLocalDate = LocalDate.of(2019, 2, 21);
```

Если дата начала совпадает с датой окончания, мы останавливаем цикл посредством метода `LocalDate.isBefore(ChronoLocalDate other)`. Указанный флаговый метод проверяет, является ли эта дата более ранней по отношению к заданной дате.

Увеличивать начальную дату на один день до конечной даты можно посредством метода `LocalDate.plusDays(long daysToAdd)`. Использование этих двух методов в цикле `for` выражается в следующем исходном коде:

```
for (LocalDate date = startLocalDate;
     date.isBefore(endLocalDate); date = date.plusDays(1)) {

    // сделать что-то с этим днем
    System.out.println(date);
}
```

Снимок выходных данных должен иметь вот такой вид:

```
2019-02-01
2019-02-02
2019-02-03
...
2019-02-20
```

Начиная с JDK 9

JDK 9 способен решить эту задачу с помощью одной строки кода. Это возможно благодаря новому методу `LocalDate.datesUntil(LocalDate endExclusive)`, который возвращает `Stream<LocalDate>` с шагом приращения в один день:

```
startLocalDate.datesUntil(endLocalDate).forEach(System.out::println);
```

Если шаг приращения должен быть выражен в днях, неделях, месяцах или годах, то следует использовать метод `LocalDate.datesUntil(LocalDate endExclusive, Period step)`. Например, шаг приращения в 1 неделю может быть задан следующим образом:

```
startLocalDate.datesUntil(endLocalDate,
    Period.ofWeeks(1)).forEach(System.out::println);
```

Результат (недели 1–8, недели 8–15) должен выглядеть следующим образом:

```
2019-02-01
2019-02-08
2019-02-15
```

74. Вычисление возраста

Вероятно, наиболее часто используемый случай разницы между двумя датами связан с вычислением возраста человека. В типичной ситуации возраст человека выражается в годах, но иногда должны быть предусмотрены месяцы и даже дни.

До JDK 8

До JDK 8 попытка обеспечить хорошее решение опиралась на класс `Calendar` и/или `SimpleDateFormat`. Это решение содержится в исходном коде, прилагаемом к данной книге.

Начиная с JDK 8

Более оптимальной идеей будет обновиться до JDK 8 и опереться на следующий ниже прямолинейный фрагмент кода:

```
LocalDate startLocalDate = LocalDate.of(1977, 11, 2);
LocalDate endLocalDate = LocalDate.now();

long years = ChronoUnit.YEARS.between(startLocalDate, endLocalDate);
```

Можно прибавить месяцы и дни к результату также легко благодаря классу `Period`:

```
Period periodBetween = Period.between(startLocalDate, endLocalDate);
```

Теперь возраст в годах, месяцах и днях можно получить посредством методов `periodBetween.getYears()`, `periodBetween.getMonths()` и `periodBetween.getDays()`.

Например, между текущей датой, 28 февраля 2019 года, и 2 ноября 1977 года у нас получается 41 год, 3 месяца и 26 дней.

75. Начало и конец дня

В JDK 8 попытка отыскать начало/конец дня может быть осуществлена несколькими способами.

Рассмотрим день, выраженный посредством объекта `LocalDate`:

```
LocalDate localDate = LocalDate.of(2019, 2, 28);
```

Решение задачи нахождения начала дня 28 февраля 2019 года опирается на метод `atStartOfDay()`. Указанный метод возвращает время `LocalDateTime` с этой даты в момент полуночи, 00:00:

```
// 2019-02-28T00:00
```

```
LocalDateTime ldDayStart = localDate.atStartOfDay();
```

В качестве альтернативы это решение может использовать метод `of(LocalDate date, LocalTime time)`. Указанный метод совмещает заданную дату и время в `LocalDateTime`. Таким образом, если прошедшее время равно `LocalTime.MIN` (время полуночи в начале дня), то результат будет следующим:

```
// 2019-02-28T00:00
```

```
LocalDateTime ldDayStart = LocalDateTime.of(localDate, LocalTime.MIN);
```

Конец дня объекта `LocalDate` можно получить, используя по крайней мере два варианта решения. Один из вариантов решения состоит в том, чтобы опереться на метод `LocalDate.atTime(LocalTime time)`. Результирующий объект `LocalDateTime` может представлять комбинацию этой даты с концом дня, если решение в качестве аргумента передает `LocalTime.MAX` (время незадолго до полуночи в конце дня):

```
// 2019-02-28T23:59:59.999999999
```

```
LocalDateTime ldDayEnd = localDate.atTime(LocalTime.MAX);
```

В качестве альтернативы этот вариант решения может совмещать `LocalTime.MAX` с заданной датой посредством метода `atDate(LocalDate date)`:

```
// 2019-02-28T23:59:59.999999999
```

```
LocalDateTime ldDayEnd = LocalTime.MAX.atDate(localDate);
```

Поскольку класс `LocalDate` не знает о часовом поясе, приведенные выше примеры подвержены проблемам, вызванным разными крайними случаями, например с переходом на летнее/зимнее время. Некоторые переходы на летнее/зимнее время вво-

дят смену часа в полночь (00:00 становится 01:00 утра), что означает, что начало дня приходится на 01:00:00, а не на 00:00:00. В целях устранения этих трудностей рассмотрим примеры, расширяющие предыдущие примеры за счет использования класса `ZonedDateTime`, который осведомлен о переходе на летнее/зимнее время:

```
// 2019-02-28T00:00+08:00[Australia/Perth]
ZonedDateTime ldDayStartZone
    = localDate.atStartOfDay(ZoneId.of("Australia/Perth"));

// 2019-02-28T00:00+08:00[Australia/Perth]
ZonedDateTime ldDayStartZone = LocalDateTime
    .of(localDate, LocalTime.MIN).atZone(ZoneId.of("Australia/Perth"));

// 2019-02-28T23:59:59.999999999+08:00[Australia/Perth]
ZonedDateTime ldDayEndZone = localDate.atTime(LocalTime.MAX)
    .atZone(ZoneId.of("Australia/Perth"));

// 2019-02-28T23:59:59.999999999+08:00[Australia/Perth]
ZonedDateTime ldDayEndZone = LocalTime.MAX.atDate(localDate)
    .atZone(ZoneId.of("Australia/Perth"));
```

Теперь давайте рассмотрим следующее — `LocalDateTime`, 28 февраля 2019 года, 18:00:00:

```
LocalDateTime localDateTime = LocalDateTime.of(2019, 2, 28, 18, 0, 0);
```

Очевидный вариант решения состоит в извлечении `LocalDate` из `LocalDateTime` и применении упомянутых выше подходов. Еще один вариант решения основан на том факте, что каждая имплементация интерфейса `Temporal` (включая `LocalDate`) может воспользоваться методом `with(TemporalField field, long newValue)`. В общих чертах, метод `with()` возвращает копию этой даты с заданным полем `ChronoField` со значением `newValue`. Таким образом, если этот вариант решения устанавливает `ChronoField.NANO_OF_DAY` (наносекунды дня) как `LocalTime.MIN`, то результатом будет начало дня. Хитрость в том, чтобы конвертировать `LocalTime.MIN` в наносекунды посредством метода `toNanoOfDay()`:

```
// 2019-02-28T00:00
LocalDateTime ldtDayStart = localDateTime
    .with(ChronoField.NANO_OF_DAY, LocalTime.MIN.toNanoOfDay());
```

Это эквивалентно следующему:

```
LocalDateTime ldtDayStart = localDateTime.with(ChronoField.HOUR_OF_DAY, 0);
```

Конец дня довольно похож. Просто вместо `LocalTime.MIN` нужно передать `LocalTime.MAX`:

```
// 2019-02-28T23:59:59.999999999
LocalDateTime ldtDayEnd = localDateTime
    .with(ChronoField.NANO_OF_DAY, LocalTime.MAX.toNanoOfDay());
```


Это эквивалентно следующему:

```
LocalDateTime ldtDayEnd = localDateTime.with(  
    ChronoField.NANO_OF_DAY, 8639999999999L);
```

Как и объект `LocalDate`, объект `LocalDateTime` не знает о часовых поясах. В этом случае поможет класс `ZonedDateTime`:

```
// 2019-02-28T00:00+08:00[Australia/Perth]  
ZonedDateTime ldtDayStartZone = localDateTime  
    .with(ChronoField.NANO_OF_DAY, LocalTime.MIN.toNanoOfDay())  
    .atZone(ZoneId.of("Australia/Perth"));
```

```
// 2019-02-28T23:59:59.999999999+08:00[Australia/Perth]  
ZonedDateTime ldtDayEndZone = localDateTime  
    .with(ChronoField.NANO_OF_DAY, LocalTime.MAX.toNanoOfDay())  
    .atZone(ZoneId.of("Australia/Perth"));
```

В качестве бонуса давайте здесь подумаем о начале/конце дня в UTC. Помимо варианта решения, опирающегося на метод `with()`, еще один вариант решения может опираться на метод `toLocalDate()` таким образом:

```
// например, 2019-02-28T09:23:10.603572Z  
ZonedDateTime zdt = ZonedDateTime.now(ZoneOffset.UTC);  
  
// 2019-02-28T00:00Z  
ZonedDateTime dayStartZdt = zdt.toLocalDate().atStartOfDay(zdt.getZone());  
  
// 2019-02-28T23:59:59.999999999Z  
ZonedDateTime dayEndZdt = zdt.toLocalDate()  
    .atTime(LocalTime.MAX).atZone(zdt.getZone());
```



Из-за многочисленных проблем с классами `java.util.Date` и `Calendar` рекомендуется избегать попыток имплементировать решение этой задачи с их помощью.

76. Разница между двумя датами

Операция вычисления разницы между двумя датами встречается очень часто (например, см. разд. 74 "Вычисление возраста"). Давайте рассмотрим набор других подходов, которые можно использовать для получения разницы между двумя датами в миллисекундах, секундах, часах и т. д.

До JDK 8

Рекомендуется представлять информацию о дате и времени посредством классов `java.util.Date` и `Calendar`. Самая простая разница вычисляется выраженной в миллисекундах. Этот вариант решения содержится в исходном коде, прилагаемом к данной книге.

Начиная с JDK 8

Начиная с JDK 8, рекомендуемым способом представления информации о дате и времени является интерфейс `Temporal` (например, имплементирующие его классы — `DateTime`, `DateLocalTime`, `ZonedDateTime` и т. д.).

Допустим, что у нас есть два объекта `LocalDate` — 1 января 2018 года и 1 марта 2019 года:

```
LocalDate ld1 = LocalDate.of(2018, 1, 1);
LocalDate ld2 = LocalDate.of(2019, 3, 1);
```

Самый простой способ вычислить разницу между этими двумя объектами `Temporal` — использовать класс `ChronoUnit`. Помимо представления стандартного набора единиц периодов дат, класс `ChronoUnit` идет в комплекте с несколькими удобными методами, в том числе с методом `between(Temporal t1Inclusive, Temporal t2Exclusive)`. Как следует из его названия, метод `between()` вычисляет промежуток времени между двумя объектами `Temporal`. Давайте посмотрим, как это работает для вычисления разницы между `ld1` и `ld2` в днях, месяцах и годах:

```
// 424
long betweenInDays = Math.abs(ChronoUnit.DAYS.between(ld1, ld2));

// 14
long betweenInMonths = Math.abs(ChronoUnit.MONTHS.between(ld1, ld2));

// 1
long betweenInYears = Math.abs(ChronoUnit.YEARS.between(ld1, ld2));
```

В качестве альтернативы каждый `Temporal` выставляет наружу метод с именем `until()`. На самом деле `LocalDate` имеет их два, один из которых возвращает значение типа `Period` как разницу между двумя датами, а другой — значение типа `long` как разницу между двумя датами в указанной единице времени. Использование метода, который возвращает значение типа `Period`, выглядит следующим образом:

```
Period period = ld1.until(ld2);

// Разница как Period: 1y2m0d
System.out.println("Difference as Period: " + period.getYears() + "y"
    + period.getMonths() + "m" + period.getDays() + "d");
```

Использование метода, который позволяет нам указывать единицу времени, выглядит следующим образом:

```
// 424
long untilInDays = Math.abs(ld1.until(ld2, ChronoUnit.DAYS));

// 14
long untilInMonths = Math.abs(ld1.until(ld2, ChronoUnit.MONTHS));

// 1
long untilInYears = Math.abs(ld1.until(ld2, ChronoUnit.YEARS));
```

Метод `ChronoUnit.convert()` также полезен в случае `LocalDateTime`. Рассмотрим два объекта `LocalDateTime` — 1 января 2019 года 22:15:15 и 1 марта 2019 года 23:15:15:

```
LocalDateTime ldt1 = LocalDateTime.of(2019, 1, 1, 22, 15, 15);
LocalDateTime ldt2 = LocalDateTime.of(2019, 1, 1, 23, 15, 15);
```

Теперь давайте посмотрим на разницу между `ldt1` и `ldt2`, выраженную в минутах:

```
// 60
long betweenInMinutesWithoutZone =
    Math.abs(ChronoUnit.MINUTES.between(ldt1, ldt2));
```

и разницу, выраженную в часах, посредством метода `LocalDateTime.until()`:

```
// 1
long untilInMinutesWithoutZone = Math.abs(ldt1.until(ldt2, ChronoUnit.HOURS));
```

Но действительно удивительная вещь относительно методов `ChronoUnit.between()` и `until()` связана с тем фактом, что они работают с классом `ZonedDateTime`. Например, рассмотрим `ldt1` в часовом поясе Европа/Бухарест и в часовом поясе Австралия/Перт плюс один час:

```
ZonedDateTime zdt1 = ldt1.atZone(ZoneId.of("Europe/Bucharest"));
ZonedDateTime zdt2 = zdt1.withZoneSameInstant(
    ZoneId.of("Australia/Perth")).plusHours(1);
```

Теперь давайте применим метод `ChronoUnit.between()`, чтобы выразить разницу между `zdt1` и `zdt2` в минутах, и `ZonedDateTime.until()`, чтобы выразить разницу между `zdt1` и `zdt2` в часах:

```
// 60
long betweenInMinutesWithZone =
    Math.abs(ChronoUnit.MINUTES.between(zdt1, zdt2));
```

```
// 1
long untilInHoursWithZone = Math.abs(zdt1.until(zdt2, ChronoUnit.HOURS));
```

Наконец, давайте повторим этот технический прием, но для двух независимых объектов `ZonedDateTime`; один получен для `ldt1` и другой — для `ldt2`:

```
ZonedDateTime zdt1 = ldt1.atZone(ZoneId.of("Europe/Bucharest"));
ZonedDateTime zdt2 = ldt2.atZone(ZoneId.of("Australia/Perth"));
```

```
// 300
long betweenInMinutesWithZone =
    Math.abs(ChronoUnit.MINUTES.between(zdt1, zdt2));

// 5
long untilInHoursWithZone = Math.abs(zdt1.until(zdt2, ChronoUnit.HOURS));
```

77. Имплементация шахматных часов

Начиная с JDK 8, пакет `java.time` имеет абстрактный класс `Clock`. Главная цель этого класса — позволить нам подключать разные часы, когда это необходимо (на-

пример, для тестирования). По умолчанию Java идет в комплекте с четырьмя имплементациями: `SystemClock`, `OffsetClock`, `TickClock` и `FixedClock`. Для каждой из этих имплементаций в классе `Clock` имеются статические методы. Например, следующий ниже фрагмент кода создает `FixedClock` (часы, которые всегда возвращают один и тот же объект `Instant`):

```
Clock fixedClock = Clock.fixed(Instant.now(), ZoneOffset.UTC);
```

Существует также `TickClock`, который возвращает текущий момент `Instant`, тикающий в полных секундах для заданного часового пояса:

```
lock tickClock = Clock.tickSeconds(ZoneId.of("Europe/Bucharest"));
```



Существует также метод `tickMinutes()`, который можно использовать для тикания в полных минутах, и обобщенный метод `tick()`, который позволяет нам указать продолжительность `Duration`.

Класс `Clock` может также поддерживать часовые пояса и поясные смещения, но наиболее важным методом класса `Clock` является метод `instant()`. Указанный метод возвращает временной момент часов:

```
// 2019-03-01T13:29:34Z
```

```
System.out.println(tickClock.instant());
```



Существует также метод `millis()`, который возвращает текущий момент часов в миллисекундах.

Допустим, что мы хотим имплементировать часы, которые действуют как шахматные часы (рис. 3.3).

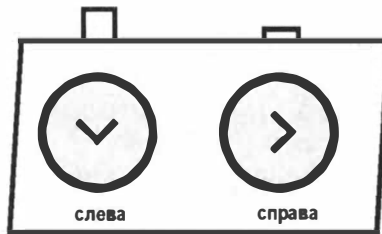


Рис. 3.3

Для того чтобы имплементировать класс `Clock`, необходимо выполнить несколько шагов:

1. Расширить класс `Clock`.
2. Имплементировать интерфейс `Serializable`.
3. Переопределить, по крайней мере, абстрактные методы, унаследованные от класса `Clock`.

Скелет класса ChessClock выглядит следующим образом:

```
public class ChessClock extends Clock implements Serializable {
    @Override
    public ZoneId getZone() {
        ...
    }

    @Override
    public Clock withZone(ZoneId zone) {
        ...
    }

    @Override
    public Instant instant() {
        ...
    }
}
```

Наш класс `ChessClock` работает только с UTC; никакой другой часовой пояс поддерживаться не будет. Это означает, что методы `getZone()` и `withZone()` могут быть имплементированы следующим образом (конечно, их имплементация может быть изменена в будущем):

```
@Override
public ZoneId getZone() {
    return ZoneOffset.UTC;
}

@Override
public Clock withZone(ZoneId zone) {
    throw new UnsupportedOperationException(
        "Класс ChessClock работает только в часовом поясе UTC");
}
```

Кульминацией нашей имплементации является метод `instant()`. Сложность заключается в управлении двумя объектами `Instant`: один для игрока слева (`instantLeft`) и другой для игрока справа (`instantRight`). Мы можем связать каждый вызов метода `instant()` с тем фактом, что текущий игрок сделал ход, и теперь очередь другого игрока. Таким образом, эта логика говорит о том, что один и тот же игрок не может вызывать метод `instant()` дважды. Имплементируя эту логику, метод `instant()` выглядит следующим образом:

```
public class ChessClock extends Clock implements Serializable {
    public enum Player {
        LEFT,
        RIGHT
    }
}
```

```
private static final long serialVersionUID = 1L;

private Instant instantStart;
private Instant instantLeft;
private Instant instantRight;
private long timeLeft;
private long timeRight;
private Player player;

public ChessClock(Player player) {
    this.player = player;
}

public Instant gameStart() {
    if (this.instantStart == null) {
        this.timeLeft = 0;
        this.timeRight = 0;
        this.instantStart = Instant.now();
        this.instantLeft = instantStart;
        this.instantRight = instantStart;

        return instantStart;
    }

    throw new IllegalStateException(
        "Игра уже началась. Остановите ее и попробуйте еще раз.");
}

public Instant gameEnd() {
    if (this.instantStart != null) {
        instantStart = null;

        return Instant.now();
    }

    throw new IllegalStateException("Игра еще не началась.");
}

@Override
public ZoneId getZone() {
    return ZoneOffset.UTC;
}

@Override
public Clock withZone(ZoneId zone) {
```

```
        throw new UnsupportedOperationException(
            "Класс ChessClock работает только в часовом поясе UTC");
    }

    @Override
    public Instant instant() {
        if (this.instantStart != null) {
            if (player == Player.LEFT) {
                player = Player.RIGHT;

                long secondsLeft = Instant.now().getEpochSecond()
                    - instantRight.getEpochSecond();
                instantLeft = instantLeft.plusSeconds(secondsLeft - timeLeft);
                timeLeft = secondsLeft;

                return instantLeft;
            } else {
                player = Player.LEFT;

                long secondsRight = Instant.now().getEpochSecond()
                    - instantLeft.getEpochSecond();
                instantRight = instantRight.plusSeconds(secondsRight - timeRight);
                timeRight = secondsRight;

                return instantRight;
            }
        }

        throw new IllegalStateException("Игра еще не началась.");
    }
}
```

Таким образом, в зависимости от того, какой игрок вызывает метод `instant()`, исходный код вычисляет количество секунд, необходимых этому игроку, для того чтобы он успел подумать, пока не выполнит ход. Более того, исходный код переключает игрока, поэтому следующий вызов `instant()` будет иметь дело с другим игроком.

Рассмотрим шахматную партию, начинающуюся 2019-03-01T14:02:46.309459Z:

```
ChessClock chessClock = new ChessClock(Player.LEFT);

// 2019-03-01T14:02:46.309459Z
Instant start = chessClock.gameStart();
```

Далее игроки выполняют последовательность ходов, пока игрок справа не выигрывает:

```
Левый пошел первым по прошествии 2 секунд: 2019-03-01T14:02:48.309459Z
Правый пошел по прошествии 5 секунд: 2019-03-01T14:02:51.309459Z
Левый пошел по прошествии 6 секунд: 2019-03-01T14:02:54.309459Z
Правый пошел по прошествии 1 секунд: 2019-03-01T14:02:52.309459Z
Левый пошел по прошествии 2 секунд: 2019-03-01T14:02:56.309459Z
Правый пошел по прошествии 3 секунд: 2019-03-01T14:02:55.309459Z
Левый пошел по прошествии 10 секунд: 2019-03-01T14:03:06.309459Z
Правый пошел по прошествии 11 секунд and win: 2019-03-01T14:03:06.309459Z
```

Похоже, шахматные часы правильно зарегистрировали ходы игроков.

Наконец, игра заканчивается через 40 секунд:

```
Игра закончилась:2019-03-01T14:03:26.350749300Z
Instant end = chessClock.gameEnd();
```

```
Продолжительность игры: 40 секунд
// Duration.between(start, end).getSeconds();
```

Резюме

Миссия выполнена! В настоящей главе представлен всесторонний обзор работы с информацией о дате и времени. Данными такого рода манипулируют приложения широкого спектра. Поэтому наличие решений приведенных выше задач в вашем арсенале является просто обязательным. Все классы, начиная с `Date` и `Calendar` и заканчивая `LocalDate`, `LocalTime`, `LocalDateTime`, `ZoneDateTime`, `OffsetDateTime`, `OffsetTime` и `Instant`, важны и очень полезны в ежедневной работе с участием даты и времени.

Вы можете скачать приложения из этой главы, чтобы увидеть результаты и получить дополнительные сведения, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

4

Логический вывод типов

Эта глава содержит 21 задачу с привлечением JEP 286, или логического вывода типа локальной переменной в Java (local variable type inference, LVTI), типа так называемой переменной `var`. Эти задачи были тщательно проработаны, в них раскрыты лучшие практические приемы и сделаны акценты на часто встречающиеся ошибки, связанные с использованием `var`.

К концу этой главы вы узнаете все, что вам нужно знать о типе `var`, для того чтобы применять его на практике.

Задачи

Используйте следующие задачи для проверки навыков программирования логического вывода типов. Перед тем как обратиться к решениям и скачать примеры программ, настоятельно рекомендуется изучить каждую задачу.

78. **Простой пример с использованием переменной типа `var`.** Написать программу, которая иллюстрирует правильное использование логического вывода типа (переменной типа `var`) в отношении удобочитаемости кода.
79. **Использование типа `var` с примитивными типами.** Написать программу, которая иллюстрирует применение переменной `var` с примитивными типами Java (`int`, `long`, `float` и `double`).
80. **Использование типа `var` и неявного приведения типов для поддержания технической сопровождаемости исходного кода.** Написать программу, которая иллюстрирует, как переменная `var` и неявное приведение типов могут поддерживать техническую сопровождаемость исходного кода.
81. **Явное понижающее приведение типов или как избежать типа `var`.** Написать программу, которая иллюстрирует сочетание типа `var` и явного понижающего приведения типов и объясняет причину, почему тип `var` следует избегать.

82. **Как отказаться от типа `var`, если вызываемые имена не содержат для людей достаточной информации о типе.** Привести примеры, где типа `var` следует избегать, потому что его сочетание с вызываемыми именами приводит к тому, что люди теряют информацию.
83. **Сочетание LVTI и программирования согласно интерфейсу.** Написать программу, которая иллюстрирует использование типа `var` посредством *программирования согласно интерфейсу*.
84. **Сочетание LVTI и ромбовидного оператора.** Написать программу, которая иллюстрирует использование типа `var` с *ромбовидным* оператором.
85. **Присвоение массива типа `var`.** Написать программу, которая передает массив переменной `var`.
86. **Использование LVTI в составных объявлениях.** Объяснить и проиллюстрировать использование LVTI с составными объявлениями.
87. **LVTI и область видимости переменной.** Объяснить и проиллюстрировать причину, по которой LVTI должно максимально уменьшать область видимости переменной.
88. **LVTI и тернарный оператор.** Написать несколько фрагментов кода, которые иллюстрируют преимущества сочетания LVTI и тернарного оператора.
89. **LVTI и циклы `for`.** Написать несколько примеров, иллюстрирующих использование LVTI в циклах `for`.
90. **LVTI и потоки.** Написать несколько фрагментов кода, которые иллюстрируют использование LVTI и потоков Java.
91. **Использование LVTI для разбиения вложенных/крупных цепочек выражений.** Написать программу, которая демонстрирует использование LVTI для разбиения вложенной/крупной цепочки выражений.
92. **LVTI и возвращаемый и параметрический типы метода.** Написать несколько фрагментов кода, которые иллюстрируют использование LVTI и методов Java в терминах возвращаемых и параметрических типов.
93. **LVTI и анонимные классы.** Написать несколько фрагментов кода, которые иллюстрируют использование LVTI в анонимных классах.
94. **LVTI может быть *финальной* и *фактически финальной*.** Написать несколько фрагментов кода, которые иллюстрируют, как LVTI может использоваться для финальных и практически финальных переменных.
95. **LVTI и лямбда-выражения.** Объяснить на нескольких фрагментах кода, как LVTI может использоваться в сочетании с лямбда-выражениями.
96. **LVTI и инициализаторы `null`, экземплярные переменные и переменные блоков `catch`.** Объяснить на примерах, как LVTI может использоваться в сочетании с *инициализаторами* `null`, экземплярными переменными и блоками `catch`.

97. **LVTI и обобщенные типы, Т.** Написать несколько фрагментов кода, которые иллюстрируют, как LVTI может использоваться в сочетании с обобщенными типами.
98. **LVTI, подстановочные знаки, коварианты и контраварианты.** Написать несколько фрагментов кода, которые иллюстрируют, как LVTI может использоваться в сочетании с подстановочными знаками, ковариантами и контравариантами.

Решения

В следующих разделах описаны решения перечисленных выше задач. Помните, что обычно не существует одного единственно правильного варианта решения той или иной задачи. Кроме того, имейте в виду, что приведенные здесь объяснения включают только самые интересные и важные детали, необходимые для решения. Вы можете скачать примеры решений, чтобы увидеть дополнительные детали и поэкспериментировать с программами, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

78. Простой пример с использованием переменной типа `var`

Начиная с версии 10, Java идет в комплекте с JEP 286, или Java LVTI, также именуемым типом `var`.



Идентификатор `var` — это не ключевое слово языка Java, а зарезервированное имя типа.

Это 100%-ное языковое средство времени компиляции без каких-либо побочных эффектов с точки зрения байт-кода, времени выполнения или производительности. Одним словом, логический вывод типа локальной переменной (LVTI) применяется к локальным переменным и работает следующим образом: компилятор проверяет правую сторону и логически выводит реальный тип (если правая сторона является инициализатором, то он использует этот тип).



Указанное языковое средство обеспечивает безопасность времени компиляции. Это означает, что мы не можем скомпилировать приложение, которое пытается выполнить неправильное присваивание. Если компилятор определил конкретный/фактический тип переменной `var`, то мы можем передавать переменным значения только этого типа.

Логический вывод типа локальной переменной имеет несколько преимуществ. Например, он снижает многословность кода и уменьшает избыточность и стереотипность кода. Более того, благодаря логическому выводу типа локальной переменной

сокращается время, затрачиваемое на написание кода, в особенности в случаях, связанных с обильными объявлениями, как показано ниже:

```
// без var
Map<Boolean, List<Integer>> evenAndOddMap...
```

```
// с var
var evenAndOddMap = ..
```

Спорным преимуществом является удобочитаемость кода. Некоторые утверждают, что использование `var` снижает удобочитаемость кода, в то время как другие поддерживают противоположное мнение. В зависимости от варианта использования, оно может приводить к снижению удобочитаемости, но истина заключается в том, что, как правило, мы много внимания уделяем осмысленным именам полей (экземплярным переменным) и пренебрегаем именами локальных переменных. Для примера рассмотрим следующий метод:

```
public Object fetchTransferableData(String data)
    throws UnsupportedOperationException, IOException {

    StringSelection ss = new StringSelection(data);
    DataFlavor[] df = ss.getTransferDataFlavors();
    Object obj = ss.getTransferData(df[0]);

    return obj;
}
```

Это короткий метод, он имеет осмысленное имя и чистую имплементацию. Но взгляните на имена локальных переменных. Их имена резко сокращены (и представляют собой просто аббревиатуры), но это не проблема, т. к. левая сторона инструкций предоставляет достаточно информации, для того чтобы суметь легко понять тип каждой локальной переменной. Теперь давайте напишем этот код с использованием логического вывода типа локальной переменной (LVTI):

```
public Object fetchTransferableData(String data)
    throws UnsupportedOperationException, IOException {

    var ss = new StringSelection(data);
    var df = ss.getTransferDataFlavors();
    var obj = ss.getTransferData(df[0]);

    return obj;
}
```

Очевидно, что удобочитаемость кода снизилась, т. к. теперь труднее логически вывести тип локальных переменных. Как показывает снимок экрана на рис. 4.1, компилятор не имеет проблем с логическим выводом правильных типов, но для людей он намного сложнее для понимания.

Декомпиляция класса, содержащего этот метод

```

public Object fetchTransferableData(String data)
    throws UnsupportedOperationException, IOException {

    StringSelection ss = new StringSelection(data);
    DataFlavor[] df = ss.getTransferDataFlavors();
    Object obj = ss.getTransferData(df[0]);

    return obj;
}

```

Рис. 4.1

Решение проблемы состоит в предоставлении осмысленного имени локальным переменным при использовании LVTI. Например, исходный код может восстановить свою удобочитаемость, если имена локальных переменных будут указаны следующим образом:

```

public Object fetchTransferableData(String data)
    throws UnsupportedOperationException, IOException {

    var stringSelection = new StringSelection(data);
    var dataFlavorsArray = stringSelection.getTransferDataFlavors();
    var obj = stringSelection.getTransferData(dataFlavorsArray[0]);

    return obj;
}

```

Проблема удобочитаемости также обусловлена тем, что в стандартной ситуации мы склонны рассматривать тип как первичную информацию, а имя переменной — как вторичную, тогда как это должно быть наоборот.

Давайте посмотрим еще на два примера, которые призваны обеспечить соблюдение вышеупомянутых положений. Метод, использующий коллекции (например, коллекцию List), выглядит следующим образом:

```

// Избегать
public List<Player> fetchPlayersByTournament(String tournament) {
    var t = tournamentRepository.findByName(tournament);
    var p = t.getPlayers();

    return p;
}

// Предпочитать
public List<Player> fetchPlayersByTournament(String tournament) {
    var tournamentName = tournamentRepository.findByName(tournament);
    var playerList = tournamentName.getPlayers();

    return playerList;
}

```

Предоставление осмысленных имен для локальных переменных не означает, что вы впадаете в избыточное именование переменных.

Например, следует избегать именования переменных, просто повторяя имя типа:

```
// Избегать
var fileCacheImageOutputStream
    = new FileCacheImageOutputStream(..., ...);

// Предпочитать
var outputStream = new FileCacheImageOutputStream(..., ...);

// Или
var outputStreamOfFoo = new FileCacheImageOutputStream(..., ...);
```

79. Использование типа *var* с примитивными типами

Проблема использования LVTI с примитивными типами (`int`, `long`, `float` и `double`) заключается в том, что ожидаемые и логически выводимые типы могут различаться. Очевидно, что это приводит к путанице и неожиданному поведению в коде.

Виновной стороной в этой ситуации является *неявное приведение типов*, используемое типом `var`.

Например, рассмотрим следующие два объявления, которые основаны на явных примитивных типах:

```
boolean valid = true;    // эта переменная имеет тип boolean
char c = 'c';           // эта переменная имеет тип char
```

Теперь давайте заменим явный примитивный тип на LVTI:

```
var valid = true;        // логически выводится как boolean
var c = 'c';             // логически выводится как char
```

Великолепно! Пока никаких проблем нет! Теперь рассмотрим другой набор объявлений, основанных на явных примитивных типах:

```
int intNumber = 10;      // эта переменная имеет тип int
long longNumber = 10;    // эта переменная имеет тип long
float floatNumber = 10;  // эта переменная имеет тип float, 10.0
double doubleNumber = 10; // эта переменная имеет тип double, 10.0
```

Давайте последуем логике из первого примера и заменим явные примитивные типы на LVTI:

```
// Избегать
var intNumber = 10;      // логически выводится как int
var longNumber = 10;    // логически выводится как int
var floatNumber = 10;   // логически выводится как int
var doubleNumber = 10;  // логически выводится как int
```

Декомпиляция класса, содержащего эти объявления

```
int intNumber = 10;
int longNumber = 10;
int floatNumber = 10;
int doubleNumber = 10;
```

Рис. 4.2

В соответствии с приведенным на рис. 4.2 снимком экрана тип всех четырех переменных был логически выведен как целочисленный.

Решение этой проблемы состоит в использовании *явных литералов Java*:

```
// Предпочитать
var intNumber = 10;           // логически выводится как int
var longNumber = 10L;        // логически выводится как long
var floatNumber = 10F;       // логически выводится как float, 10.0
var doubleNumber = 10D;      // логически выводится как double, 10.0
```

Наконец, рассмотрим случай числа с десятичными знаками, как вот здесь:

```
var floatNumber = 10.5;      // логически выводится как double
```

Имя переменной намекает на то, что 10.5 имеет тип `float`, но на самом деле ее тип логически выводится как `double`. Таким образом, рекомендуется опираться на *литералы* даже для чисел с десятичными знаками (в особенности для чисел типа `float`):

```
var floatNumber = 10.5F;     // логически выводится как float
```

80. Использование типа `var` и неявного приведения типов для поддержания технической сопровождаемости исходного кода

В предыдущем разд. "79. Использование типа `var` с примитивными типами" мы увидели, что сочетание типа `var` с неявным приведением типов может становиться причиной реальных проблем. Но в некоторых сценариях это сочетание бывает выгодным и поддерживает техническую сопровождаемость исходного кода.

Рассмотрим следующий сценарий: нам нужно написать метод, который располагается между двумя существующими методами внешнего API с именем `ShoppingAddicted` (по экстраполяции эти методы могут быть двумя веб-службами, конечными точками и т. д.). Один из методов посвящен возврату наилучшей цены для заданной корзины покупок. В сущности, этот метод берет несколько продуктов и делает запрос в различные интернет-магазины, чтобы получить наилучшую цену.

Результирующая цена возвращается как значение типа `int`. Заготовка этого метода представлена следующим образом:

```
public static int fetchBestPrice(String[] products) {
    float realprice = 399.99F; // запрос цен в магазинах
    int price = (int) realprice;

    return price;
}
```

Другой метод получает цену как значение типа `int` и выполняет платеж. Если платеж проходит успешно, то этот метод возвращает `true`:

```
public static boolean debitCard(int amount) {
    return true;
}
```

Теперь, программируя по отношению к этому коду, наш метод будет действовать как клиент следующим образом (клиенты могут принимать решения по поводу того, какие товары покупать, и наш код будет возвращать им наилучшую цену и соответственно дебетовать их платежные карты):

```
// Избегать
public static boolean purchaseCart(long customerId) {
    int price = ShoppingAddicted.fetchBestPrice(new String[0]);
    boolean paid = ShoppingAddicted.debitCard(price);

    return paid;
}
```

Но через некоторое время владельцы API `ShoppingAddicted` понимают, что они теряют деньги, конвертируя реальную цену в значение типа `int` (например, реальная цена составляет 399,99, но в форме типа `int` она составляет 399,0, что означает потерю 99 центов). Поэтому они решают отказаться от этой практики и возвращать реальную цену как значение типа `float`:

```
public static float fetchBestPrice(String[] products) {
    float realprice = 399.99F; // запрос цен в магазинах

    return realprice;
}
```

Поскольку возвращенная цена имеет тип `float`, метод `debitCard()` тоже обновляется:

```
public static boolean debitCard(float amount) {
    return true;
}
```

Но после того, как мы перейдем к новому релизу API `ShoppingAddicted`, код завершится сбоем с возможными исключениями, связанными с потерями при *конвертировании из типа* `float` в тип `int`. Это нормально, т. к. наш код ожидает тип `int`. Поскольку наш исходный код плохо переносит эти изменения, он должен быть изменен соответствующим образом.

Тем не менее, если мы бы предвидели эту ситуацию и вместо `int` использовали бы `var`, то код работал бы без проблем благодаря *неявному приведению типов*:

```
// Предпочитать
public static boolean purchaseCart(long customerId) {
    var price = ShoppingAddicted.fetchBestPrice(new String[0]);
    var paid = ShoppingAddicted.debitCard(price);

    return paid;
}
```


81. Явное понижающее приведение типов или как избежать типа `var`

В разд. 79 "*Использование типа `var` с примитивными типами*" мы говорили об использовании литералов с примитивными типами (`int`, `long`, `float` и `double`) во избежание проблем, вызванных с неявным приведением типов. Но не все примитивные типы Java могут пользоваться преимуществами литералов. В такой ситуации лучше всего избегать использования переменной типа `var`. Но давайте посмотрим почему!

Взгляните на следующие объявления переменных типа `byte` и `short`:

```
byte byteNumber = 25;      // эта переменная имеет тип byte
short shortNumber = 1463; // эта переменная имеет тип short
```

Если мы заменим явные типы типом `var`, логически будет выведен тип `int`:

```
var byteNumber = 25;      // логически выводится как int
var shortNumber = 1463;   // логически выводится как int
```

К сожалению, *литералов* для этих двух примитивных типов не существует. Единственный способ помочь компилятору логически вывести правильные типы — опереться на явное понижающее приведение типов:

```
var byteNumber = (byte) 25;      // логически выводится как byte
var shortNumber = (short) 1463; // логически выводится как short
```

Хотя этот код успешно компилируется и работает надлежащим образом, мы не можем сказать, что использование переменной типа `var` принесло какую-либо ценность по сравнению с использованием явных типов. Так что, в данном случае, лучше избегать типа `var` и явного понижающего приведения типов.

82. Как отказаться от типа `var`, если вызываемые имена не содержат для людей достаточной информации о типе

Дело в том, что тип `var` вовсе не является серебряной пулей, и данная задача еще раз это подчеркнет. Следующий фрагмент кода может быть написан с использованием явных типов либо типа `var` без потери информации:

```
// используя явные типы
MemoryCacheImageInputStream is = new MemoryCacheImageInputStream(...);
JavaCompiler jc = ToolProvider.getSystemJavaCompiler();
StandardJavaFileManager fm = compiler.getStandardFileManager(...);
```

Таким образом, мигрирование показанного выше фрагмента кода в тип `var` приведет к следующему ниже коду (имена переменных были выбраны визуальным осмотром вызываемых имен с правой стороны):

```
// используя var
var inputStream = new MemoryCacheImageInputStream(...);
var compiler = ToolProvider.getSystemJavaCompiler();
var fileManager = compiler.getStandardFileManager(...);
```

То же самое произойдет и в крайнем случае избыточного именования:

```
// используя var
var inputStreamOfCachedImages = new MemoryCacheImageInputStream(...);
var javaCompiler = ToolProvider.getSystemJavaCompiler();
var standardFileManager = compiler.getStandardFileManager(...);
```

Таким образом, приведенный выше код не поднимает никаких проблем в выборе имен переменных и в удобочитаемости. Вызываемые *имена* содержат достаточно информации, для того чтобы люди чувствовали себя комфортно с типом `var`.

Но давайте рассмотрим следующий ниже фрагмент кода:

```
// Избегать
public File fetchBinContent() {
    return new File(...);
}

// вызывается из другого места
// обратите внимание на имя переменной, bin
var bin = fetchBinContent();
```

Людям довольно трудно логически вывести тип, возвращаемый вызываемым именем, не проверив возвращаемый тип этого *имени*, `fetchBinContent()`. В качестве общего правила следует уяснить, что в таких случаях решение задачи должно избегать типа `var` и опираться на явные типы, поскольку в правой части недостаточно информации для выбора правильного имени переменной и получения хорошо читаемого кода:

```
// вызывается из другого места
// теперь левая сторона содержит достаточно информации
File bin = fetchBinContent();
```

Таким образом, если тип `var` в сочетании с вызываемыми *именами* приводит к потере ясности, то лучше избегать употребления типа `var`. Игнорирование этого положения может привести к путанице и увеличит время, необходимое для понимания и/или расширения кода.

Рассмотрим еще один пример, основанный на классе `java.nio.channels.Selector`. Этот класс выставляет наружу статический метод с именем `open()`, который возвращает только что открытый `Selector`. Но если мы захватим этот возврат в переменной, объявленной с помощью типа `var`, заманчиво думать, что этот метод возвращает значение типа `boolean`, представляющее успешное открытие текущего селектора. Использование типа `var` без учета возможной потери ясности приводит именно к таким проблемам. Всего несколько таких проблем, как эта, и исходный код станет настоящей головной болью.

83. Сочетание LVTI и программирования согласно интерфейсу

Передовой опыт программирования на Java побуждает нас к тому, чтобы привязывать код к абстракции. Другими словами, мы должны опираться на технический прием *программирования согласно интерфейсу*¹.

Указанный прием очень хорошо подходит для объявлений коллекций. Например, рекомендуется объявлять список `ArrayList` следующим образом:

```
List<String> players = new ArrayList<>();
```

Мы также должны избегать чего-то подобного:

```
ArrayList<String> players = new ArrayList<>();
```

По аналогии с первым примером исходный код создает экземпляр класса `ArrayList` (или класса `HashSet`, `HashMap` и т. д.), но объявляет переменную типа `List` (или типа `Set`, `Map` и т. д.). Поскольку `List`, `Set`, `Map` и многие другие являются интерфейсами (или контрактами), очень легко заменить этот экземпляр другой имплементацией интерфейса `List` (`Set` и `Map`) без последующих модификаций кода.

К сожалению, LVTI не может воспользоваться преимуществами *программирования согласно интерфейсу*. Другими словами, когда мы используем тип `var`, логически выводимым типом является конкретная имплементация, а не контракт. Например, замена `List<String>` на `var` приведет к логически выведенному типу `ArrayList<String>`:

```
// логически выводится как ArrayList<String>
var playerList = new ArrayList<String>();
```

Тем не менее есть несколько объяснений, которые поддерживают это поведение:

- ◆ LVTI действует на локальном уровне (локальные переменные), где программирование согласно интерфейсу используется меньше, чем параметры метода либо возвращаемые из метода типы значений или типы полей;
- ◆ поскольку локальные переменные имеют малую область видимости, изменения, индуцируемые переключением на другую имплементацию, также должны быть малыми. Имплементация переключения должна оказывать малое влияние на обнаружение и исправление кода;
- ◆ LVTI рассматривает код с правой стороны в качестве инициализатора, который полезен для вывода фактического типа. Если этот инициализатор будет изменен в будущем, то логически выводимый тип может отличаться, и это станет причиной проблемой в коде, использующем эту переменную.

¹ Здесь имеется в виду программирование согласно спецификации или контракту. — Прим. перев.

84. Сочетание LVTI и ромбовидного оператора

В качестве общего правила надо принять, что LVTI в сочетании с ромбовидным оператором может давать неожиданные логически выводимые типы, если информация, необходимая для логического вывода ожидаемого типа, отсутствует в правой части.

До JDK 7, т. е. согласно проекту Coin, список `List<String>` объявлялся следующим образом:

```
List<String> players = new ArrayList<String>();
```

В приведенном выше примере явно указывается параметрический тип instantiation обобщенного класса. Начиная с JDK 7, проект Coin ввел *ромбовидный* оператор, который способен логически выводить параметрический тип instantiation обобщенного класса следующим образом:

```
List<String> players = new ArrayList<>();
```

Теперь, если мы рассмотрим этот пример с точки зрения LVTI, то получим такой результат:

```
var playerList = new ArrayList<>();
```

Но какой тип теперь будет выведен логически? Дело в том, что здесь у нас проблема, потому что вместо типа `ArrayList<String>` будет выведен тип `ArrayList<Object>`. Объяснение этому вполне очевидно: информация, необходимая для логического вывода ожидаемого типа (`String`), отсутствует (обратите внимание, что в тексте в правой части не указан тип `String`). Это дает LVTI команду логически вывести наиболее широкий применимый тип, которым в данном случае является тип `Object`.

Но если тип `ArrayList<Object>` не был типом, который мы предполагали, то нам нужно найти решение этой проблемы. Оно состоит в том, чтобы предоставить информацию, необходимую для логического вывода ожидаемого типа, следующим образом:

```
var playerList = new ArrayList<String>();
```

Теперь логически выводиться тип `ArrayList<String>`. Тип также может быть выведен косвенно. Вгляните на следующий пример:

```
var playerStack = new ArrayDeque<String>();
```

```
// логически выводиться как ArrayList<String>
```

```
var playerList = new ArrayList<>(playerStack);
```

Он также может быть выведен косвенно следующим образом:

```
Player p1 = new Player();
```

```
Player p2 = new Player();
```

```
var listOfPlayer = List.of(p1, p2); // логически выводиться как List<Player>
```

```
// Не делать этого!
```

```
var listOfPlayer = new ArrayList<>(); // логически выводиться
```

```
// как ArrayList<Object>
```

```
listOfPlayer.add(p1);
```

```
listOfPlayer.add(p2);
```

85. Присвоение массива переменной типа *var*

В качестве общего правила следует принять, что передача массива переменной типа *var* не требует скобок, []. Определить массив типа *int* посредством соответствующего явного типа можно так:

```
int[] numbers = new int[10];
```

```
// либо, что менее предпочтительно
int numbers[] = new int[10];
```

Теперь попытка интуитивно понять, как использовать тип *var* вместо типа *int*, может привести к следующим вариантам:

```
var[] numberArray = new int[10];
var numberArray[] = new int[10];
```

К сожалению, ни один из этих двух вариантов не компилируется. Решение задачи требует, чтобы мы убрали скобки с левой стороны:

```
// Предпочитать
var numberArray = new int[10]; // логически выводится как массив типа int, int[]
numberArray[0] = 3;           // работает
numberArray[0] = 3.2;         // не работает
numbers[0] = "3";             // не работает
```

На практике часто встречается инициализация массива во время объявления, как показано ниже:

```
// явный тип работает, как ожидалось
int[] numbers = {1, 2, 3};
```

Однако попытка использовать переменную типа *var* работать не будет (не компилируется):

```
// Не компилируется
var numberArray = {1, 2, 3};
var numberArray[] = {1, 2, 3};
var[] numberArray = {1, 2, 3};
```

Этот код не компилируется, потому что правая часть инструкций не имеет собственного типа.

86. Использование LVTI в составных объявлениях

Составное объявление позволяет нам объявлять группу переменных одного и того же типа, не повторяя тип. Тип указывается один раз, а переменные разделяются запятой:

```
// используя явный тип
String pending = "ожидает", processed = "обработан", deleted = "удален";
```

Замена типа *String* на тип *var* приведет к тому, что код не будет компилироваться:

```
// Не компилируется
var pending = "ожидает", processed = "обработан", deleted = "удален";
```

Решение этой проблемы заключается в трансформировании составного объявления в одно объявление на одну строку кода:

```
// используя var, выводится тип String
var pending = "ожидает";
var processed = "обработан";
var deleted = "удален";
```

Таким образом, в качестве общего правила нужно уяснить, что LVTI не может использоваться в составных объявлениях.

87. LVTI и область видимости переменной

Рекомендации по чистому коду включают поддержание малой области видимости для всех локальных переменных. Это одно из золотых правил чистого кода, которому следовали еще до появления LVTI.

Это правило поддерживает фазу считывания и отладки, а также ускоряет процесс поиска ошибок и написания исправлений. Рассмотрим следующий пример, который нарушает это правило:

```
// Избегать
...
var stack = new Stack<String>();
stack.push("Джон");
stack.push("Мартин");
stack.push("Анхел");
stack.push("Кристиан");

// 50 строк кода, который не использует стек
// Джон, Мартин, Анхел, Кристиан
stack.forEach(...);
```

Приведенный выше код объявляет стек с четырьмя именами, содержит 50 строк кода, которые не используют этот стек, и завершается обходом этого стека в цикле посредством метода `forEach()`. Этот метод унаследован от `java.util.Vector` и будет обходить стек в цикле, как и любой вектор (Джон, Мартин, Анхел, Кристиан). Это тот порядок прохождения, который нам нужен.

Но позже мы решаем переключиться со стека на очередь `ArrayDeque` (причина не имеет значения). На этот раз метод `forEach()` будет предусмотрен классом `ArrayDeque`. Поведение этого метода отличается от метода `Vector.forEach()` тем, что цикл будет перебирать элементы, следуя дисциплине обхода элементов "последним вошел, первым вышел" (last in first out, LIFO) (Кристиан, Анхел, Мартин, Джон):

```
// Избегать
...
var stack = new ArrayDeque<String>();
stack.push("Джон");
```

```
stack.push("Мартин");
stack.push("Анхел");
stack.push("Кристиан");
```

```
// 50 строк кода, который не использует стек
// Кристиан, Анхел, Мартин, Джон
stack.forEach(...);
```

Это не входило в наши намерения! Мы переключились на `ArrayDeque` для других целей, а не для того, чтобы повлиять на порядок обхода в цикле. Но увидеть в коде дефект довольно трудно, т. к. часть кода, содержащая фрагмент `forEach()`, не находится в непосредственной близости от кода, где мы внесли модификации (50 строк ниже этой строки кода). В наших обязанностях придумать решение, которое максимизирует шансы исправить этот дефект быстро и избегает кучу прокруток кода вверх и вниз в целях понять, что происходит. Решение состоит в следовании правилу чистого кода, которое мы упомянули ранее, и написании этого кода с малой областью видимости для переменной `stack`:

```
// Предпочитать
...
var stack = new Stack<String>();
stack.push("Джон");
stack.push("Мартин");
stack.push("Анхел");
stack.push("Кристиан");
```

```
// Джон, Мартин, Анхел, Кристиан
stack.forEach(...);
```

```
// 50 строк кода, который не использует стек
```

Теперь, когда мы переключаемся со стека `Stack` на очередь `ArrayQueue`, мы должны быстрее заметить ошибку и сможем ее устранить.

88. LVTI и тернарный оператор

При условии, что *тернарный* оператор написан правильно, он позволяет нам использовать разные типы операндов в правой части. Например, следующий ниже фрагмент кода не компилируется:

```
// Не компилируется
List evensOrOdds = containsEven ?
    List.of(10, 2, 12) : Set.of(13, 1, 11);
```

```
// Не компилируется
Set evensOrOdds = containsEven ?
    List.of(10, 2, 12) : Set.of(13, 1, 11);
```

Тем не менее этот код можно исправить, переписав его с использованием правильных/поддерживаемых явных типов:

```
Collection evensOrOdds = containsEven ?
    List.of(10, 2, 12) : Set.of(13, 1, 11);
```

```
Object evensOrOdds = containsEven ?
    List.of(10, 2, 12) : Set.of(13, 1, 11);
```

Похожая попытка будет неудачной для следующего фрагмента кода:

```
// Не компилируется
int numberOrText = intOrString ? 2234 : "2234";

// Не компилируется
String numberOrText = intOrString ? 2234 : "2234";
```

Однако это можно исправить так:

```
Serializable numberOrText = intOrString ? 2234 : "2234";

Object numberOrText = intOrString ? 2234 : "2234";
```

Таким образом, для того чтобы иметь *тернарный* оператор с разными типами операндов в правой части, разработчик должен сочетать правильный тип, который поддерживает обе условные ветви. В качестве альтернативы разработчик может опереться на LVTI, как показано ниже (разумеется, это работает и для одинаковых типов операндов):

```
// логически выводимый тип, Collection<Integer>
var evensOrOddsCollection = containsEven ?
    List.of(10, 2, 12) : Set.of(13, 1, 11);

// логически выводимый тип, Serializable
var numberOrText = intOrString ? 2234 : "2234";
```

Из этих примеров не следует делать заключение о том, что тип переменной `var` логически выводится во время выполнения! Это не так!

89. LVTI и циклы *for*

Операция объявления простых циклов `for` с использованием явных типов является тривиальной, как показано ниже:

```
// явный тип
for (int i = 0; i < 5; i++) {
    ...
}
```

В качестве альтернативы мы можем применить расширенный цикл `for`:

```
List<Player> players = List.of(
    new Player(), new Player(), new Player());
```



```
for (Player player: players) {  
    ...  
}
```

Начиная с JDK 10, мы можем заменить явные типы переменных `i` и `player` на тип `var` следующим образом:

```
for (var i = 0; i < 5; i++) { // i логически выводится как тип int  
    ...  
}
```

```
for (var player: players) { // i логически выводится как тип Player  
    ...  
}
```

Использование типа `var` бывает полезным, когда тип обходимого в цикле массива, коллекции и других конструкций изменяется. Например, с помощью типа `var` обе версии представленного ниже массива могут быть пройдены в цикле без указания явного типа:

```
// переменная 'array', представляющая массив int[]  
int[] array = { 1, 2, 3 };  
  
// либо та же переменная 'array', но представляющая String[]  
String[] array = {  
    "1", "2", "3"  
};  
  
// в зависимости от того, как определяется 'array',  
// 'i' будет логически выведено как int либо как тип String  
for (var i: array) {  
    System.out.println(i);  
}
```

90. LVTI и потоки

Рассмотрим следующий ниже поток `Stream<Integer>`:

```
// явный тип  
Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5);  
numbers.filter(t -> t % 2 == 0).forEach(System.out::println);
```

Использовать LVTI вместо `Stream<Integer>` довольно просто. Надо лишь заменить тип `Stream<Integer>` на тип `var`:

```
// с помощью var логически выводится как тип Stream<Integer>  
var numberStream = Stream.of(1, 2, 3, 4, 5);  
numberStream.filter(t -> t % 2 == 0).forEach(System.out::println);
```

Вот еще один пример:

```
// явные типы  
Stream<String> paths = Files.lines(Path.of("..."));  
List<File> files = paths.map(p -> new File(p)).collect(toList());
```

```
// с помощью var
// логически выводится как тип Stream<String>
var pathStream = Files.lines(Path.of(""));

// логически выводится как тип List<File>
var fileList = pathStream.map(p -> new File(p)).collect(toList());
```

Похоже, что Java 10 (и его LVTI) и Java 8 (и его API потоков) составляют хорошую команду.

91. Использование LVTI для разбиения вложенных/крупных цепочек выражений

Крупные/вложенные выражения — это, как правило, такие фрагменты кода, которые выглядят довольно внушительно и отпугивающе. Они обычно рассматриваются как части *заумного* или *изошренного* кода. Можно поспорить на счет того, что это делать хорошо или плохо, но, скорее всего, баланс склоняется в пользу тех, кто утверждает, что такого кода следует избегать. Например, взгляните на следующее выражение:

```
List<Integer> ints = List.of(1, 1, 2, 3, 4, 4, 6, 2, 1, 5, 4, 5);

// Избегать
int result = ints.stream()
    .collect(Collectors.partitioningBy(i -> i % 2 == 0))
    .values()
    .stream()
    .max(Comparator.comparing(List::size))
    .orElse(Collections.emptyList())
    .stream()
    .mapToInt(Integer::intValue)
    .sum();
```

Такие выражения могут быть написаны преднамеренно, либо они могут представлять собой финальный результат поступательного процесса, который обогащает первоначально небольшое выражение во времени. Тем не менее, когда такие выражения начинают становиться пробелами в удобочитаемости, их нужно разбить на части посредством локальных переменных. Но такая работа не приносит удовольствия и может считаться изнурительной неприятностью, которую мы хотим избежать:

```
List<Integer> ints = List.of(1, 1, 2, 3, 4, 4, 6, 2, 1, 5, 4, 5);

// Предпочитать
Collection<List<Integer>> evenAndOdd = ints.stream()
    .collect(Collectors.partitioningBy(i -> i % 2 == 0))
    .values();
```

```
List<Integer> evenOrOdd = evenAndOdd.stream()
    .max(Comparator.comparing(List::size))
    .orElse(Collections.emptyList());
```

```
int sumEvenOrOdd = evenOrOdd.stream()
    .mapToInt(Integer::intValue)
    .sum();
```

Взгляните на типы локальных переменных в приведенном выше коде. Мы имеем `Collection<List<Integer>>`, `List<Integer>` и `int`. Очевидно, что для извлечения и написания этих явных типов требуется некоторое время. Это может быть уважительной причиной для того, что не разбивать данное выражение на части. Тем не менее тривиальность использования типа `var` вместо явных типов является соблазнительной, если мы хотим принять стиль локальной переменной, потому что он экономит время, которое обычно тратится на извлечение явных типов:

```
var intList = List.of(1, 1, 2, 3, 4, 4, 6, 2, 1, 5, 4, 5);
```

```
// Предпочитать
var evenAndOdd = intList.stream()
    .collect(Collectors.partitioningBy(i -> i % 2 == 0))
    .values();
```

```
var evenOrOdd = evenAndOdd.stream()
    .max(Comparator.comparing(List::size))
    .orElse(Collections.emptyList());
```

```
var sumEvenOrOdd = evenOrOdd.stream()
    .mapToInt(Integer::intValue)
    .sum();
```

Потрясающе! Теперь задача компилятора — определить типы этих локальных переменных. Мы выбираем только те точки, где разбиваем выражение, и разграничиваем их с помощью типа `var`.

92. LVTI и возвращаемый и аргументный типы метода

LVTI нельзя использовать в качестве возвращаемого или аргументного типа метода; вместо этого переменные типа `var` могут передаваться как аргументы метода или хранить возвращаемый метод (в инструкции `return`). Давайте рассмотрим эти положения на нескольких примерах.

♦ LVTI нельзя использовать в качестве возвращаемого типа из метода, следующий ниже код не компилируется:

```
// Не компилируется
public var fetchReport(Player player, Date timestamp) {
    return new Report();
}
```

- ◆ LVТИ нельзя использовать в качестве аргументного типа внутри объявления метода, следующий ниже код не компилируется:

```
public Report fetchReport(var player, var timestamp) {
    return new Report();
}
```

- ◆ Переменные типа `var` могут передаваться в качестве аргументов методов или хранить возвращаемый результат метода, следующий ниже код успешно компилируется и работает:

```
public Report checkPlayer() {
    var player = new Player();
    var timestamp = new Date();
    var report = fetchReport(player, timestamp);

    return report;
}

public Report fetchReport(Player player, Date timestamp) {
    return new Report();
}
```

93. LVТИ и анонимные классы

LVТИ можно использовать для анонимных классов. Давайте взглянем на следующий пример анонимного класса, который использует явный тип для переменной `weighter`:

```
public interface Weighter {
    int getWeight(Player player);
}

Weighter weighter = new Weighter() {
    @Override
    public int getWeight(Player player) {
        return ...;
    }
};
```

```
Player player = ...;
int weight = weighter.getWeight(player);
```

Теперь посмотрим, что произойдет, если мы применим LVТИ:

```
var weighter = new Weighter() {
    @Override
    public int getWeight(Player player) {
        return ...;
    }
};
```

94. LVTI может быть финальным и практически финальным

В качестве быстрого напоминания, начиная с Java SE 8, локальный класс может обращаться к локальным переменным и параметрам охватывающего блока, которые являются финальными или практически финальными. Переменная или параметр, значение которых никогда не изменяется после их инициализации, являются практически финальными.

Следующий ниже фрагмент кода представляет вариант использования *практически финальной* переменной (попытка повторно присвоить переменной `ratio` значение приведет к ошибке, а это означает, что данная переменная является *практически финальной*) и двух финальных переменных (попытка повторно присвоить переменным `limit` и `bmi` значение приведет к ошибке, и значит, эти переменные являются финальными):

```
public interface Weighter {
    float getMarginOfError();
}

float ratio = fetchRatio(); // эта переменная является практически финальной

var weighter = new Weighter() {
    @Override
    public float getMarginOfError() {
        return ratio * ...;
    }
};

ratio = fetchRatio(); // это повторное присвоение станет причиной ошибки

public float fetchRatio() {
    final float limit = new Random().nextFloat(); // финальная переменная
    final float bmi = 0.00023f;                // финальная переменная

    limit = 0.002f; // это повторное присвоение станет причиной ошибки
    bmi = 0.25f;   // это повторное присвоение станет причиной ошибки

    return limit * bmi / 100.12f;
}
```

Теперь давайте заменим явные типы на тип `var`. Компилятор логически выведет для этих переменных (`ratio`, `limit` и `bmi`) правильные типы и будет поддерживать их состояние — переменная `ratio` будет *практически финальной*, тогда как переменные `limit` и `bmi` — финальными.

Попытка повторно присвоить любой из них значение вызовет специфическую ошибку:

var ratio = fetchRatio(); // эта переменная является практически финальной

```
var weighter = new Weighter() {
    @Override
    public float getMarginOfError() {
        return ratio * ...;
    }
};
```

ratio = fetchRatio(); // это повторное присвоение станет причиной ошибки

```
public float fetchRatio() {
    final var limit = new Random().nextFloat(); // финальная переменная
    final var bmi = 0.00023f; // финальная переменная

    limit = 0.002f; // это повторное присвоение станет причиной ошибки
    bmi = 0.25f; // это повторное присвоение станет причиной ошибки

    return limit * bmi / 100.12f;
}
```

95. LVTI и лямбда-выражения

Проблема с использованием LVTI и лямбда-выражений заключается в том, что конкретный тип не получается вывести логически. Лямбда-выражения и *инициализаторы* ссылок на методы не допускаются. Это положение является частью ограничений типа `var`; поэтому лямбда-выражения и ссылки на методы требуют явных целевых типов.

Например, следующий ниже фрагмент кода не компилируется:

```
// Не компилируется.
// Лямбда-выражение нуждается в явном целевом типе
var incrementX = x -> x + 1;
```

```
// ссылка на метод требует явного целевого типа
var exceptionIAE = IllegalArgumentException::new;
```

Поскольку тип `var` нельзя использовать, оба фрагмента кода необходимо написать следующим образом:

```
Function<Integer, Integer> incrementX = x -> x + 1;
```

```
Supplier<IllegalArgumentException> exceptionIAE
    = IllegalArgumentException::new;
```

Но в контексте лямбда-выражений Java 11 позволяет использовать тип `var` в параметрах лямбд. Например, в Java 11 следующий ниже код работает (более подробную информацию можно найти в описании "JEP 323: синтаксис локальных переменных для параметров лямбд", расположенном по адресу <https://openjdk.java.net/jeps/323>):

```
@FunctionalInterface
public interface Square {
    int calculate(int x);
}
```

```
Square square = (var x) -> x * x;
```

Однако имейте в виду, что следующий код работать не будет:

```
var square = (var x) -> x * x; // логически вывести не получается
```

96. LVTI и инициализаторы *null*, экземплярные переменные и переменные блоков *catch*

Что общего у LVTI с *инициализаторами null*, экземплярными переменными и переменными блоков *catch*? Дело в том, что LVTI нельзя использовать ни с одним из них. Следующие ниже попытки окажутся безуспешными.

◆ LVTI нельзя использовать с *инициализаторами null*:

```
// выдаст ошибку типа: инициализатор переменной равен 'null'
var message = null;
// выдаст: не получается использовать 'var' на переменной без инициализатора
var message;
```

◆ LVTI нельзя использовать с экземплярными переменными (полями):

```
public class Player {

    private var age;    // ошибка: 'var' здесь не допустимо
    private var name;  // ошибка: 'var' здесь не допустимо
    ...
}
```

◆ LVTI нельзя использовать в переменных блока *catch*:

```
try {
    TimeUnit.NANOSECONDS.sleep(1000);
} catch (var ex) { ... }
```

Инструкция *try* с объявлением ресурса

С другой стороны, переменная типа `var` очень хорошо подходит для инструкции `try` с *объявлением ресурса* (или ресурсов), как в следующем ниже примере:

```
// явный тип
try (PrintWriter writer = new PrintWriter(new File("welcome.txt"))) {
```

```

writer.println("Welcome message");
}

// используя тип var
try (var writer = new PrintWriter(new File("welcome.txt"))) {
    writer.println("Welcome message");
}

```

97. LVTI и обобщенные типы T

Для того чтобы понять, как LVTI может сочетаться с обобщенными типами, давайте начнем с примера. Следующий метод является классическим вариантом использования обобщенного типа T:

```

public static <T extends Number> T add(T t) {
    T temp = t;
    ...
    return temp;
}

```

В этом случае мы можем заменить T на var, и код будет работать нормально:

```

public static <T extends Number> T add(T t) {
    var temp = t;
    ...
    return temp;
}

```

Таким образом, локальные переменные, имеющие обобщенные типы, могут использовать преимущества LVTI. Давайте рассмотрим другие примеры, сначала на основе обобщенного типа T:

```

public <T extends Number> T add(T t) {
    List<T> numberList = new ArrayList<T>();
    numberList.add(t);
    numberList.add((T) Integer.valueOf(3));
    numberList.add((T) Double.valueOf(3.9));

    // ошибка: несравнимые типы, String не получается конвертировать в T
    // numbers.add("5");

    return numberList.get(0);
}

```

Теперь давайте заменим тип List<T> переменной типа var:

```

public <T extends Number> T add(T t) {
    var numberList = new ArrayList<T>();
    numberList.add(t);
    numberList.add((T) Integer.valueOf(3));
    numberList.add((T) Double.valueOf(3.9));
}

```



```
// ошибка: несравнимые типы, String не получается конвертировать в T
// numbers.add("5");

return numberList.get(0);
}
```

Обратите внимание на экземпляр списка `ArrayList` и на наличие у него типа `T`. Этого делать не следует (он будет логически выведен как `ArrayList<Object>` и будет игнорировать реальный тип позади обобщенного типа, `T`):

```
var numberList = new ArrayList<>();
```

98. LVTI, подстановочные знаки, коварианты и контраварианты

Работа по замене подстановочных знаков, ковариантов и контравариантов на LVTI является незаурядной, и ее следует выполнять с полным осознанием последствий.

LVTI и подстановочные знаки

Прежде всего, давайте поговорим о LVTI и подстановочных знаках (?). На практике широко принято ассоциировать подстановочные знаки с `Class` и писать что-то вроде этого:

```
// явный тип
Class<?> clazz = Long.class;
```

В таких случаях нет никаких проблем с использованием переменной типа `var` вместо `Class<?>`. В зависимости от типа правой стороны компилятор логически выведет правильный тип. В этом примере компилятор выведет `Class<Long>`.

Но обратите внимание на то, что замена подстановочных знаков на LVTI должна делаться осторожно и вы должны быть осведомлены о последствиях (или побочных эффектах). Давайте взглянем на пример, в котором решение заменить подстановочный знак переменной типа `var` является плохим. Рассмотрим следующий фрагмент кода:

```
Collection<?> stuff = new ArrayList<>();
stuff.add("hello"); // ошибка времени компиляции
stuff.add("world"); // ошибка времени компиляции
```

Этот код не компилируется из-за несовместимых типов. Очень плохим подходом было бы исправить этот код, заменив подстановочный знак переменной типа `var`, как показано ниже:

```
var stuff = new ArrayList<>();
strings.add("hello"); // ошибки нет
strings.add("world"); // ошибки нет
```

При использовании переменной типа `var` ошибка исчезнет, но это не то, что мы имели в виду, когда писали приведенный выше код (код с ошибками несовмести-

мости типов). Таким образом, в качестве общего правила, не следует заменять `Foo<?>` на `var` просто потому, что некоторые досадные ошибки исчезнут по волшебству! Попробуйте подумать о том, какая поставлена задача, и действовать соответственно. Например, может быть, в предыдущем фрагменте кода мы пытались определить `ArrayList<String>` и по ошибке получили `Collection<?>`.

LVTI и коварианты/контраварианты

Подход с заменой ковариантов (`Foo<? extends T>`) или контравариантов (`Foo<? super T>`) на LVTI является опасным, и его следует избегать.

Взгляните на следующий ниже фрагмент кода:

```
// явные типы
Class<? extends Number> intNumber = Integer.class;
Class<? super FilterReader> fileReader = Reader.class;
```

В коварианте у нас есть верхняя граница, представленная классом `Number`, тогда как в контраварианте мы имеем нижнюю границу, представленную классом `FilterReader`. Имея эти границы (или ограничения) на своем месте, следующий ниже код вызовет специфическую ошибку времени компиляции:

```
// Не компилируется
// ошибка: не получается конвертировать Class<Reader>
// в Class<? extends Number>
Class<? extends Number> intNumber = Reader.class;

// ошибка: не получается конвертировать Class<Integer>
// в Class<? super FilterReader>
Class<? super FilterReader> fileReader = Integer.class;
```

Теперь, давайте применим переменную типа `var` вместо показанных выше коварианта и контраварианта:

```
// используя var
var intNumber = Integer.class;
var fileReader = Reader.class;
```

Этот код не вызовет никаких проблем. Теперь мы можем присвоить этим переменным любой класс, и наши границы/ограничения исчезнут. Это не то, что мы собирались сделать:

```
// это компилируется без проблем
var intNumber = Reader.class;
var fileReader = Integer.class;
```

Таким образом, решение использовать переменную типа `var` в место наших коварианта и контраварианта было плохим!

Резюме

Это была последняя задача настоящей главы. Для получения дополнительной информации обратитесь к описаниям "JEP 323: синтаксис локальной переменной для лямбда-параметров" (<https://openjdk.java.net/jeps/323>) и "JEP 301: расширенные подклассы класса перечислений `Enum`" (<http://openjdk.java.net/jeps/301>). Принятие этих средств на вооружение должно пройти довольно гладко, если вы знакомы с задачами и проблемами, которые были рассмотрены в этой главе.

Вы можете скачать приложения из этой главы, чтобы увидеть результаты и получить дополнительные детали, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

5

Массивы, коллекции и структуры данных

Эта глава содержит 30 задач с привлечением массивов, коллекций и нескольких структур данных. Ее цель состоит в том, чтобы обеспечить решения категории задач, возникающих в широком спектре приложений, включая сортировку, поиск, сравнение, упорядочивание, инвертирование, заполнение, слияние, копирование и замену. Предусмотренные решения имплементированы в Java 8–12 и могут также использоваться в качестве основы для решения других родственных задач.

К концу этой главы у вас будет солидная база знаний, которая пригодится для решения разнообразных задач с привлечением массивов, коллекций и структур данных.

Задачи

Используйте следующие задачи для проверки вашего умения программировать массивы, коллекции и структуры данных. Перед тем как обратиться к решениям и скачать примеры программ, настоятельно рекомендуется изучить каждую задачу.

99. **Сортировка массива.** Написать несколько программ, которые иллюстрируют разные алгоритмы сортировки массивов. Также написать программу для перетасовки массивов.
100. **Поиск элемента в массиве.** Написать несколько программ, которые иллюстрируют способ отыскания заданного элемента (примитивного и объекта) в заданном массиве. Найти индекс и/или просто проверить, что значение находится в массиве.
101. **Проверка эквивалентности или несовпадения двух массивов.** Написать программу, которая проверяет эквивалентность двух заданных массивов или их несовпадение.

102. **Сравнение двух массивов лексикографически.** Написать программу, которая сравнивает заданные массивы лексикографически.
103. **Создание потока из массива.** Написать программу, которая создает поток из заданного массива.
104. **Минимальное, максимальное и среднее значения массива.** Написать программу, которая вычисляет минимальное, максимальное и среднее значения заданного массива.
105. **Инвертирование массива.** Написать программу, которая инвертирует заданный массив.
106. **Заполнение и настройка массива.** Написать несколько примеров заполнения массива и задания всех элементов на основе генераторной функции для вычисления каждого элемента.
107. **Следующий больший элемент.** Написать программу, которая возвращает следующий больший элемент для каждого элемента массива.
108. **Изменение размера массива.** Написать программу, которая добавляет элемент в массив, увеличивая его размер на единицу. Кроме того, написать программу, которая увеличивает размер массива на заданную длину.
109. **Создание немодифицируемых/немутируемых коллекций.** Написать несколько примеров создания немодифицируемых и немутуируемых отображений `Map`.
110. **Возврат значения по умолчанию из коллекции `Map`.** Написать программу, которая получает значение из отображения `Map` либо значение по умолчанию.
111. **Вычисление отсутствия/присутствия элемента в отображении `Map`.** Написать программу, которая вычисляет значение отсутствующего ключа или новое значение текущего ключа.
112. **Удаление элемента из отображения `Map`.** Написать программу, которая удаляет элемент из отображения `Map` посредством заданного ключа.
113. **Замена элементов в отображении `Map`.** Написать программу, которая заменяет данные в отображении `Map`.
114. **Сравнение двух отображений `Map`.** Написать программу, которая сравнивает два отображения `Map`.
115. **Сортировка отображения `Map`.** Написать программу, которая сортирует отображение `Map`.
116. **Копирование отображения `HashMap`.** Написать программу, которая выполняет мелкую и глубокую копию коллекции `HashMap`.
117. **Слияние отображений `Map`.** Написать программу, которая объединяет два отображения `Map`.

118. **Удаление всех элементов коллекции, которые совпадают с предикатом.** Написать программу, которая удаляет все элементы коллекции, которые совпадают с заданным предикатом.
119. **Конвертирование коллекции в массив.** Написать программу, которая конвертирует коллекцию в массив.
120. **Фильтрация коллекции по списку.** Написать несколько решений для фильтрации коллекции по списку. Показать лучший способ сделать это.
121. **Замена элементов списка.** Написать программу, которая заменяет каждый элемент списка результатом применения к нему заданного оператора.
122. **Нитебезопасные коллекции, стеки и очереди.** Написать несколько программ, которые иллюстрируют использование нитебезопасных коллекций Java.
123. **Поиск сперва в ширину.** Написать программу, которая имплементирует алгоритм поиска сперва в ширину (BFS).
124. **Префиксное дерево.** Написать программу, которая имплементирует структуру данных — префиксное дерево (trie).
125. **Кортеж.** Написать программу, которая имплементирует структуру данных — кортеж.
126. **Структура данных Union Find.** Написать программу, которая имплементирует алгоритм объединения и поиска (Union-Find).
127. **Дерево Фенвика, или двоичное индексированное дерево.** Написать программу, которая имплементирует алгоритм дерева Фенвика.
128. **Фильтр Блума.** Написать программу, которая имплементирует алгоритм "фильтр Блума".

Решения

В следующих разделах описаны решения перечисленных выше задач. Помните, что обычно не существует одного единственно правильного варианта решения той или иной задачи. Кроме того, имейте в виду, что приведенные здесь объяснения включают только самые интересные и важные детали, необходимые для решения. Вы можете скачать примеры решений, чтобы увидеть дополнительные детали и поэкспериментировать с программами, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

99. Сортировка массива

Операция сортировки массива встречается очень часто во многих dome-нах/приложениях. Она настолько распространена, что Java предоставляет встроенное решение для сортировки массивов примитивов и объектов с помощью компа-

ратора. Это решение работает очень хорошо и является предпочтительным в большинстве случаев. Давайте рассмотрим разные решения в следующих разделах.

Решения, встроенные в JDK

Встроенное решение называется методом `sort()`, и он идет в разнообразных разновидностях в классе `java.util.Arrays` (более 15 разновидностей).

Позади метода `sort()` стоит результативный алгоритм сортировки под названием быстрой сортировки с двойным опорным элементом (`dual-pivot quicksort`).

Допустим, что нам нужно отсортировать массив целых чисел в естественном порядке (примитивный тип `int`). В этом мы можем опереться на метод `Arrays.sort(int[] a)`, как в следующем ниже примере:

```
int[] integers = new int[]{...};
Arrays.sort(integers);
```

Иногда нам нужно отсортировать массив объектов. Допустим, что у нас есть класс `Melon`:

```
public class Melon {
    private final String type;
    private final int weight;

    public Melon(String type, int weight) {
        this.type = type;
        this.weight = weight;
    }
}
```

```
// геттеры опущены для краткости
```

Массив экземпляров класса `Melon` может быть отсортирован по возрастанию веса с помощью соответствующего компаратора:

```
Melon[] melons = new Melon[] { ... };

Arrays.sort(melons, new Comparator<Melon>() {
    @Override
    public int compare(Melon melon1, Melon melon2) {
        return Integer.compare(melon1.getWeight(), melon2.getWeight());
    }
});
```

Тот же результат можно получить, переписав приведенный выше код с помощью лямбда-выражения:

```
Arrays.sort(melons, (Melon melon1, Melon melon2)
    -> Integer.compare(melon1.getWeight(), melon2.getWeight()));
```

Более того, массивы предоставляют метод для параллельной сортировки элементов, `parallelSort()`. За кулисами этого метода используется алгоритм параллельной сор-

тировки слиянием на основе пула разветвления/соединения `ForkJoinPool`, который разбивает массив на подмассивы, которые также сортируются и затем объединяются. Вот пример:

```
Arrays.parallelSort(melons, new Comparator<Melon>() {
    @Override
    public int compare(Melon melon1, Melon melon2) {
        return Integer.compare(melon1.getWeight(), melon2.getWeight());
    }
});
```

Либо то же самое посредством лямбда-выражения:

```
Arrays.parallelSort(melons, (Melon melon1, Melon melon2)
    -> Integer.compare(melon1.getWeight(), melon2.getWeight()));
```

Приведенные выше примеры сортируют массив по возрастанию, но иногда нам нужно отсортировать его по убыванию. Когда мы сортируем массив объектов и опираемся на компаратор, мы можем просто умножить результат, возвращаемый методом `Integer.compare()`, на `-1`:

```
Arrays.sort(melons, new Comparator<Melon>() {
    @Override
    public int compare(Melon melon1, Melon melon2) {
        return (-1) * Integer.compare(melon1.getWeight(),
            melon2.getWeight());
    }
});
```

Как вариант, мы можем просто поменять местами аргументы в методе `compare()`.

В случае массива примитивных типов, упакованных в объект, решение может опираться на метод `Collections.reverse()`, как в следующем примере:

```
Integer[] integers = new Integer[] {3, 1, 5};

// 1, 3, 5
Arrays.sort(integers);

// 5, 3, 1
Arrays.sort(integers, Collections.reverseOrder());
```

К сожалению, встроенное решение для сортировки массива примитивов в убывающем порядке отсутствует. Чаще всего, если мы хотим по-прежнему опираться на метод `Arrays.sort()`, то решение этой задачи состоит в инвертировании массива ($O(n)$) после того, как он отсортирован в возрастающем порядке:

```
// отсортировать по возрастанию
Arrays.sort(integers);

// инвертировать массив, получив его в убывающем порядке
for (int leftHead = 0, rightHead = integers.length - 1;
    leftHead < rightHead; leftHead++, rightHead--) {
```



```

int elem = integers[leftHead];
integers[leftHead] = integers[rightHead];
integers[rightHead] = elem;
}

```

Еще одно решение может опираться на функциональный стиль Java 8 и операцию упаковывания¹ (имейте в виду, что операция упаковывания является довольно времязатратной):

```

int[] descIntegers = Arrays.stream(integers)
    .boxed() // либо .mapToObj(i -> i)
    .sorted((i1, i2) -> Integer.compare(i2, i1))
    .mapToInt(Integer::intValue)
    .toArray();

```

Другие алгоритмы сортировки

Дело в том, что существует много других алгоритмов сортировки. Каждый из них имеет свои плюсы и минусы, и наилучший способ выбрать среди них подходящий — сравнить их с учетом ситуации, специфичной для приложения.

Давайте прозкзаменуем некоторые из них, как показано в следующем далее разделе, и начнем с довольно медленного алгоритма.

Сортировка пузырьком

Сортировка пузырьком — это простой алгоритм, который выталкивает элементы массива вверх. Это означает, что он обходит массив в цикле несколько раз и меняет местами соседние элементы, если они расположены в неправильном порядке (рис. 5.1).

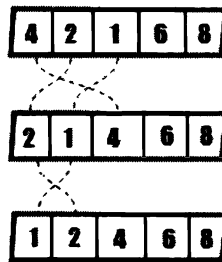


Рис. 5.1

Временная сложность алгоритма составляет: в лучшем случае — $O(n)$, в среднем случае — $O(n^2)$, в худшем случае — $O(n^2)$.

Пространственная сложность составляет в худшем случае $O(1)$.

¹ Упаковывание (boxing) — это конвертирование примитивного типа в объект. — Прим. перев.

Служебный метод, имплементирующий сортировку пузырьком, выглядит следующим образом:

```
public static void bubbleSort(int[] arr) {
    int n = arr.length;

    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {

            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Существует также его оптимизированная версия, которая опирается на цикл `while`. Вы можете найти его в исходном коде, прилагаемом к этой книге, под названием `bubbleSortOptimized()`.

В качестве сравнения производительности времени выполнения, для случайного массива из 100 000 целых чисел, оптимизированная версия будет работать примерно на 2 секунды быстрее.

Указанные выше имплементации хорошо работают для сортировки массивов примитивов, но для сортировки массива объектов нам нужно ввести в код `Comparator` следующим образом:

```
public static <T> void bubbleSortWithComparator(
    T arr[], Comparator<? super T> c) {

    int n = arr.length;

    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {

            if (c.compare(arr[j], arr[j + 1]) > 0) {
                T temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Помните приводившийся ранее класс `Melon`? Так вот, мы можем написать для него `Comparator`, имплементировав интерфейс `Comparator`:

```
public class MelonComparator implements Comparator<Melon> {
    @Override
    public int compare(Melon o1, Melon o2) {
        return o1.getType().compareTo(o2.getType());
    }
}
```

Как вариант, в функциональном стиле Java 8, мы имеем следующее:

```
// Восходящий порядок
Comparator<Melon> byType = Comparator.comparing(Melon::getType);

// Нисходящий порядок
Comparator<Melon> byType
    = Comparator.comparing(Melon::getType).reversed();
```

Имея массив объектов класса `Melon`, приведенный выше `Comparator` и метод `bubbleSortWithComparator()` в служебном классе с именем `ArraySorts`, можно написать что-то вроде следующего:

```
Melon[] melons = {...};
ArraySorts.bubbleSortWithComparator(melons, byType);
```

Оптимизированная версия сортировки пузырьком с компаратором была опущена для краткости, но она имеется в исходном коде, прилагаемом к этой книге.



Сортировка пузырьком работает быстро, когда массив почти отсортирован. Кроме того, она хорошо подходит для сортировки "кроликов" (больших элементов, которые находятся близко к началу массива) и " черепах" (малых элементов, которые находятся близко к концу массива). Но в целом этот алгоритм является медленным.

Сортировка вставками

Алгоритм сортировки вставками основан на простом процессе управления. Он начинается со второго элемента и сравнивает его с предыдущим элементом. Если предыдущий элемент больше текущего, то алгоритм меняет местами элементы. Этот процесс продолжается до тех пор, пока предыдущий элемент не станет меньше текущего элемента.

В данном случае алгоритм переходит к следующему элементу массива и повторяет процесс управления (рис. 5.2).

Временная сложность алгоритма составляет: в лучшем случае — $O(n)$, в среднем случае — $O(n^2)$, в худшем случае — $O(n^2)$.

Пространственная сложность составляет в худшем случае $O(1)$.

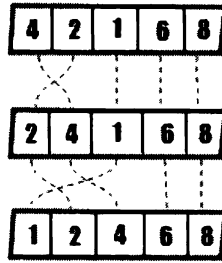


Рис. 5.2

Опираясь на этот процесс, имплементацию для примитивных типов можно записать так:

```
public static void insertionSort(int arr[]) {
    int n = arr.length;

    for (int i = 1; i < n; ++i) {

        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }

        arr[j + 1] = key;
    }
}
```

Для сравнения элементов массива объектов класса `Melon` нам необходимо ввести в имплементацию `Comparator` следующим образом:

```
public static <T> void insertionSortWithComparator(
    T arr[], Comparator<? super T> c) {

    int n = arr.length;

    for (int i = 1; i < n; ++i) {

        T key = arr[i];
        int j = i - 1;

        while (j >= 0 && c.compare(arr[j], key) > 0) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }

        arr[j + 1] = key;
    }
}
```

Здесь мы имеем компаратор, который производит сортировку дынь по сорту и весу, написанную в функциональном стиле Java 8 с использованием метода `thenComparing()`:

```
Comparator<Melon> byType = Comparator.comparing(Melon::getType)
    .thenComparing(Melon::getWeight);
```

Имея массив объектов класса `Melon`, указанный выше `Comparator` и метод `insertionSortWithComparator()` в служебном классе с именем `ArraySorts`, можно написать что-то вроде следующего:

```
Melon[] melons = {...};
ArraySorts.insertionSortWithComparator(melons, byType);
```



Этот алгоритм бывает быстрым для малых и во многом отсортированных массивов. Кроме того, он хорошо работает при добавлении в массив новых элементов. Он также является очень эффективным по потребляемой памяти, т. к. по ней перемещается один-единственный элемент.

Сортировка подсчетом

Сортировка подсчетом начинается с вычисления минимального и максимального элементов в массиве. На основе вычисленных минимума и максимума алгоритм определяет новый массив, который будет нужен для подсчета неотсортированных элементов, используя элемент в качестве индекса. Кроме того, этот новый массив модифицируется так, чтобы каждый элемент в каждом индексе хранил сумму предыдущих количеств. Наконец, отсортированный массив получается из этого нового массива.

Временная сложность алгоритма составляет: в лучшем случае — $O(n + k)$, в среднем случае — $O(n + k)$, в худшем случае — $O(n + k)$.

Пространственная сложность составляет в худшем случае $O(k)$.



- k — это число возможных значений в интервале.
- n — это число элементов, подлежащих сортировке.

Давайте рассмотрим небольшой пример. Исходный массив `arr` содержит следующие элементы: 4, 2, 6, 2, 6, 8, 5(рис. 5.3).

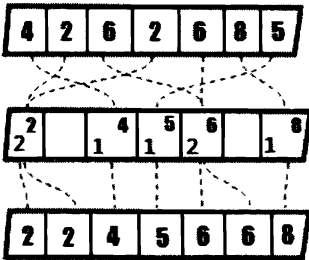


Рис. 5.3

Минимальный элемент равен 2, максимальный — 8. Новый массив `counts` будет иметь размер, равный максимуму минус минимум плюс 1: $8 - 2 + 1 = 7$.

Подсчет каждого элемента приведет к следующему массиву (`counts[arr[i] - min]++`):

`counts[2] = 1 (4); counts[0] = 2 (2); counts[4] = 2 (6);`

`counts[6] = 1 (8); counts[3] = 1 (5);`

Теперь мы должны обойти этот массив в цикле и использовать его для восстановления отсортированного массива, как в следующей имплементации:

```
public static void countingSort(int[] arr) {
    int min = arr[0];
    int max = arr[0];

    for (int i = 1; i < arr.length; i++) {
        if (arr[i] < min) {
            min = arr[i];
        } else if (arr[i] > max) {
            max = arr[i];
        }
    }

    int[] counts = new int[max - min + 1];

    for (int i = 0; i < arr.length; i++) {
        counts[arr[i] - min]++;
    }

    int sortedIndex = 0;

    for (int i = 0; i < counts.length; i++) {
        while (counts[i] > 0) {
            arr[sortedIndex++] = i + min;
            counts[i]--;
        }
    }
}
```

Этот алгоритм очень быстрый.

Сортировка кучи

Алгоритм сортировки кучи (или кучевая сортировка) опирается на двоичную кучу (полное двоичное дерево).

Временная сложность алгоритма составляет: в лучшем случае — $O(n \log n)$, в среднем случае — $O(n \log n)$, в худшем случае — $O(n \log n)$.

Пространственная сложность составляет в худшем случае $O(1)$.



Сортировка элементов в возрастающем порядке может осуществляться посредством *тах-кучи* (т. е. в которой родительский узел всегда больше или равен дочерним узлам), а в убывающем порядке — посредством *min-кучи* (т. е. в которой родительский узел всегда меньше или равен дочерним узлам).

На первом шаге данный алгоритм использует массив, предоставленный для построения этой кучи и преобразования ее в *тах-кучу* (куча представлена другим массивом). Поскольку эта куча является *тах-кучей*, самым большим элементом будет корень кучи. На следующем шаге корень заменяется последним элементом из кучи, и размер кучи уменьшается на 1 (последний узел из кучи удаляется). Элементы, находящиеся в верхней части кучи, выводятся в отсортированном порядке. Последний шаг состоит из процедуры *heapify* (рекурсивного процесса, который строит кучу сверху вниз) и корня кучи (реконструирования *тах-кучи*). Эти три шага повторяются до тех пор, пока размер кучи не превысит 1 (рис. 5.4).

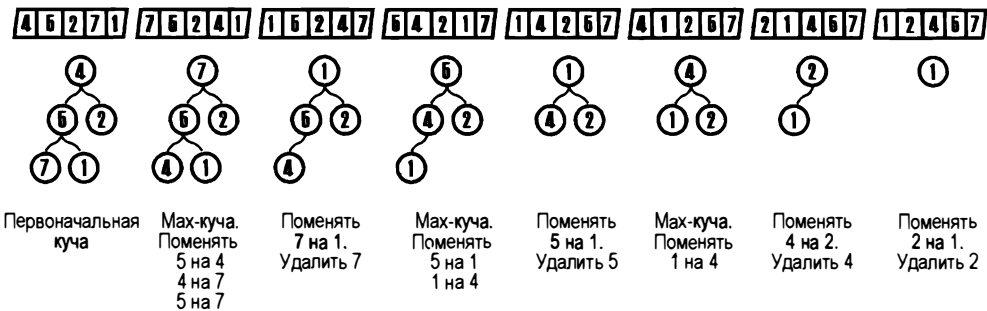


Рис. 5.4

Например, допустим, что у нас есть массив из приведенной выше схемы — 4, 5, 2, 7, 1. Тогда:

1. На первом шаге строим кучу: 4, 5, 2, 7, 1.
2. Строим *тах-кучу*: 7, 5, 2, 4, 1 (мы поменяли 5 на 4, 4 на 7 и 5 на 7).
3. Далее меняем корень (7) на последний элемент (1) и удаляем 7. Результат: 1, 5, 2, 4, 7.
4. Далее снова строим *тах-кучу*: 5, 4, 2, 1 (мы поменяли 5 на 1 и 1 на 4).
5. Меняем корень (5) на последний элемент (1) и удаляем 5. Результат: 1, 4, 2, 5, 7.
6. Затем снова строим *тах-кучу*: 4, 1, 2 (мы поменяли 1 на 4).
7. Меняем корень (4) на последний элемент (2) и удаляем 4. Результат: 2, 1.
8. Эта куча является *тах-кучей*, поэтому меняем корень (2) на последний элемент (1) и удаляем 2: 1, 2, 4, 5, 7.
9. Готово! В куче (1) остался один элемент.

В исходном коде приведенный выше пример можно обобщить следующим образом:

```
public static void heapSort(int[] arr) {
    int n = arr.length;

    buildHeap(arr, n);

    while (n > 1) {
        swap(arr, 0, n - 1);
        n--;
        heapify(arr, n, 0);
    }
}

private static void buildHeap(int[] arr, int n) {
    for (int i = arr.length / 2; i >= 0; i--) {
        heapify(arr, n, i);
    }
}

private static void heapify(int[] arr, int n, int i) {
    int left = i * 2 + 1;
    int right = i * 2 + 2;
    int greater;

    if (left < n && arr[left] > arr[i]) {
        greater = left;
    } else {
        greater = i;
    }

    if (right < n && arr[right] > arr[greater]) {
        greater = right;
    }

    if (greater != i) {
        swap(arr, i, greater);
        heapify(arr, n, greater);
    }
}

private static void swap(int[] arr, int x, int y) {
    int temp = arr[x];

    arr[x] = arr[y];
    arr[y] = temp;
}
```


Если мы хотим сравнить объекты, то должны ввести в имплементацию `Comparator`. Данное решение под названием `heapSortWithComparator()` доступно в исходном коде, прилагаемом к этой книге.

Здесь компаратор написан в функциональном стиле Java 8, который использует методы `thenComparing()` и `reversed()` для сортировки объектов класса `Melon` в убывающем порядке по сорту и весу:

```
Comparator<Melon> byType = Comparator.comparing(Melon::getType)
    .thenComparing(Melon::getWeight).reversed();
```

Имея массив объектов класса `Melon`, приведенный выше компаратор и метод `heapSortWithComparator()` в служебном классе с именем `ArraySorts`, можно написать что-то вроде следующего:

```
Melon[] melons = {...};
ArraySorts.heapSortWithComparator(melons, byType);
```



Сортировка кучи работает довольно быстро, но нестабильно. Например, сортировка массива, который уже отсортирован, может оставить его в другом порядке.

Здесь мы остановимся на нашем небольшом исследовании сортировки массивов, но в коде, прилагаемом к этой книге, имеется еще несколько алгоритмов сортировки (рис. 5.5).

```
● bubbleSort(int[] arr) void
● bubbleSortWithComparator(T[] arr, Comparator<? super T> c) void
● bubbleSortOptimized(int[] arr) void
● bubbleSortOptimizedWithComparator(T[] arr, Comparator<? super T> c) void
● bucketSort(int[] arr) void
● cocktailSort(int[] arr) void
● countingSort(int[] arr) void
● cycleSort(int[] arr) void
● exchangeSort(int[] arr) void
● heapSort(int[] arr) void
● heapSortWithComparator(T[] arr, Comparator<? super T> c) void
● insertionSort(int[] arr) void
● insertionSortWithComparator(T[] arr, Comparator<? super T> c) void
● mergeSort(int[] arr) void
● pancakeSort(int[] arr) void
● quickSort(int[] arr, int left, int right) void
● quickSortWithComparator(T[] arr, int left, int right, Comparator<? super T> c) void
● radixSort(int[] arr, int radix) void
● selectionSort(int[] arr) void
● shellSort(int[] arr) void
● shuffleInt(int[] arr) void
● shuffleObj(T[] arr) void
```

Рис. 5.5

Существует много других алгоритмов, предназначенных для сортировки массивов. Некоторые из них строятся на тех, что представлены здесь (например, сортировка расческой, коктейльная сортировка, сортировки "четное–нечетное" являются разновидностями сортировки пузырьком, корзинная сортировка является дистрибутивной сортировкой, обычно опирающейся на сортировку вставкой, поразрядная сортировка (LSD) является стабильной дистрибутивной сортировкой, похожей на корзинную сортировку, и гномья сортировка является вариацией сортировки вставками).

Другие подходы отличаются (например, быстрая сортировка имплементирована методом `Arrays.sort()`, а сортировка слиянием имплементирована методом `Arrays.parallelSort()`).

В качестве бонуса к этому разделу давайте посмотрим, как перетасовать массив. Эффективный способ достижения этой цели основан на перетасовке Фишера — Йейтса (также именуемой перетасовкой Кнута). По сути, мы обходим массив в обратном порядке и произвольно меняем элементы местами. Для примитивов (например, типа `int`) ее имплементация выглядит следующим образом:

```
public static void shuffleInt(int[] arr) {
    int index;

    Random random = new Random();

    for (int i = arr.length - 1; i > 0; i--) {
        index = random.nextInt(i + 1);
        swap(arr, index, i);
    }
}
```

В коде, прилагаемом к этой книге, также имеется имплементация перетасовки массива объектов.



Перетасовать список довольно просто посредством метода `Collections.shuffle(List<?> list)`.

100. Поиск элемента в массиве

Когда мы отыскиваем элемент в массиве, нас может интересовать индекс, в котором этот элемент расположен, или только факт присутствия элемента в массиве. Решения, представленные в этом разделе, материализованы в методах из снимка экрана на рис. 5.6.

Давайте взглянем на разные решения в следующих далее разделах.

● containsElementObjectV1 (T[] arr, T toContain)	boolean
● containsElementObjectV2 (T[] arr, T toContain, Comparator<? super T> c)	boolean
● containsElementObjectV3 (T[] arr, T toContain, Comparator<? super T> c)	boolean
● containsElementV1 (int[] arr, int toContain)	boolean
● containsElementV2 (int[] arr, int toContain)	boolean
● containsElementV3 (int[] arr, int toContain)	boolean
● findIndexOfElementObjectV1 (T[] arr, T toFind)	int
● findIndexOfElementObjectV2 (T[] arr, T toFind, Comparator<? super T> c)	int
● findIndexOfElementObjectV3 (T[] arr, T toFind, Comparator<? super T> c)	int
● findIndexOfElementV1 (int[] arr, int toFind)	int
● findIndexOfElementV2 (int[] arr, int toFind)	int

Рис. 5.6

Проверка только присутствия элемента

Допустим, что у нас есть следующий массив целых чисел:

```
int[] numbers = {4, 5, 1, 3, 7, 4, 1};
```

Поскольку этот массив состоит из значений примитивного типа, программное решение может просто обойти массив в цикле и вернуть первое появление заданного целого числа следующим образом:

```
public static boolean containsElement(int[] arr, int toContain) {
    for (int elem: arr) {
        if (elem == toContain) {
            return true;
        }
    }

    return false;
}
```

Еще один вариант решения этой задачи может опираться на методы `Arrays.binarySearch()`. Существует несколько разновидностей этого метода, но в данном случае нам нужен именно этот: `int binarySearch(int[] a, int key)`. Указанный метод будет искать заданный ключ в заданном массиве и возвращать соответствующий индекс или отрицательное значение. Единственная проблема заключается в том, что этот метод работает только для отсортированных массивов, поэтому нам нужно предварительно отсортировать массив:

```
public static boolean containsElement(int[] arr, int toContain) {
    Arrays.sort(arr);
    int index = Arrays.binarySearch(arr, toContain);

    return (index >= 0);
}
```



Если массив уже отсортирован, то указанный выше метод можно оптимизировать, удалив шаг сортировки. Кроме того, если массив отсортирован, то указанный выше метод может возвращать индекс, в котором элемент встречается в массиве, а не значение типа `boolean`. Однако если массив не отсортирован, то имейте в виду, что возвращаемый индекс соответствует отсортированному массиву, а не несортированному (исходному) массиву. Если вы не хотите сортировать исходный массив, то рекомендуется передавать этому методу клон массива. Еще одним подходом будет клонирование массива внутри этого вспомогательного метода.

В Java 8 решение задачи может опираться на функциональный стиль. Хорошим кандидатом здесь является метод `anyMatch()`. Он возвращает значение, которое сообщает о том, соответствуют ли какие-либо элементы потока предоставленному предикату. Таким образом, нам нужно только конвертировать массив в поток, как показано ниже:

```
public static boolean containsElement(int[] arr, int toContain) {
    return Arrays.stream(arr)
        .anyMatch(e -> e == toContain);
}
```

Приведенные выше примеры довольно просто адаптировать или обобщить для любого другого примитивного типа.

Теперь давайте сосредоточимся на отыскании объектов в массивах. Возьмем класс `Melon`:

```
public class Melon {
    private final String type;
    private final int weight;

    // конструктор, геттеры, equals() и hashCode() опущены для краткости
}
```

Далее рассмотрим массив объектов класса `Melon`:

```
Melon[] melons = new Melon[] {new Melon("Crenshaw", 2000),
    new Melon("Gac", 1200), new Melon("Bitter", 2200)
};
```

Теперь допустим, что мы хотим отыскать в этом массиве дыню `Гас` весом 1200 г. Решение может быть основано на методе `equals()`, который используется для определения эквивалентности двух объектов:

```
public static <T> boolean
    containsElementObject(T[] arr, T toContain) {

    for (T elem: arr) {
        if (elem.equals(toContain)) {
            return true;
        }
    }

    return false;
}
```



Точно так же мы можем опереться на метод `Arrays.asList(arr).contains(find)`. В сущности следует конвертировать массив в `List` и вызвать метод `contains()`. За кулисами этот метод использует контракт метода `equals()`.

Если этот метод находится в служебном классе `ArraySearch`, то следующий ниже вызов вернет `true`:

```
// true
boolean found = ArraySearch.containsElementObject(
    melons, new Melon("Gac", 1200));
```

Этот вариант решения прекрасно работает до тех пор, пока мы хотим опираться на контракт метода `equals()`. Но мы можем посчитать, что наша дыня присутствует в массиве, если встречаются ее имя (`Gac`) или вес (`1200 г`). В таких случаях практичнее опираться на компаратор:

```
public static <T> boolean containsElementObject(
    T[] arr, T toContain, Comparator<? super T> c) {

    for (T elem: arr) {
        if (c.compare(elem, toContain) == 0) {
            return true;
        }
    }

    return false;
}
```

Теперь `Comparator`, учитывающий только сорт дыни, можно написать следующим образом:

```
Comparator<Melon> byType = Comparator.comparing(Melon::getType);
```

Поскольку `Comparator` игнорирует вес дыни (дыня весом `1205 г` отсутствует), то следующий ниже вызов вернет `true`:

```
// true
boolean found = ArraySearch.containsElementObject(
    melons, new Melon("Gac", 1205), byType);
```

Еще один подход основан на другой разновидности метода `binarySearch()`. Класс `Arrays` предоставляет метод `binarySearch()`, который получает `Comparator`, `<T> int` `binarySearch(T[] a, T key, Comparator<? super T> c)`. Это означает, что мы можем использовать его следующим образом:

```
public static <T> boolean containsElementObject(
    T[] arr, T toContain, Comparator<? super T> c) {

    Arrays.sort(arr, c);
    int index = Arrays.binarySearch(arr, toContain, c);

    return (index >= 0);
}
```



Если исходное состояние массива должно оставаться неизменным, то рекомендуется передавать этому методу клон массива. Еще одним подходом является клонирование массива внутри этого вспомогательного метода.

Теперь компаратор, учитывающий только вес дыни, можно написать следующим образом:

```
Comparator<Melon> byWeight = Comparator.comparing(Melon::getWeight);
```

Поскольку компаратор игнорирует сорт дыни (дыня сорта Honeydew отсутствует), следующий ниже вызов вернет true:

```
// true
boolean found = ArraySearch.containsElement(
    melons, new Melon("Honeydew", 1200), byWeight);
```

Проверка только первого индекса

Для массива примитивов простейшая имплементация говорит сама за себя:

```
public static int findIndexOfElement(int[] arr, int toFind) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == toFind) {
            return i;
        }
    }
    return -1;
}
```

Опираясь на функциональный стиль Java 8, мы можем попытаться пройти массив в цикле и отфильтровать элементы, соответствующие заданному элементу. И в конце просто вернуть первый найденный элемент:

```
public static int findIndexOfElement(int[] arr, int toFind) {
    return IntStream.range(0, arr.length)
        .filter(i -> toFind == arr[i])
        .findFirst()
        .orElse(-1);
}
```

Для массива объектов существует по меньшей мере три подхода. Во-первых, мы можем опереться на контракт equals():

```
public static <T> int findIndexOfElementObject(T[] arr, T toFind) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i].equals(toFind)) {
            return i;
        }
    }
    return -1;
}
```



Схожим образом мы можем опереться на метод `Arrays.asList(arr).indexOf(find)`. В сущности следует конвертировать массив в `List` и вызвать метод `contains()`. За кулисами этот метод использует контракт `equals()`.

Во-вторых, мы можем опереться на компаратор:

```
public static <T> int findIndexOfElementObject(
    T[] arr, T toFind, Comparator<? super T> c) {

    for (int i = 0; i < arr.length; i++) {
        if (c.compare(arr[i], toFind) == 0) {
            return i;
        }
    }
    return -1;
}
```

И в-третьих, мы можем опереться на функциональный стиль Java 8 и компаратор:

```
public static <T> int findIndexOfElementObject(
    T[] arr, T toFind, Comparator<? super T> c) {

    return IntStream.range(0, arr.length)
        .filter(i -> c.compare(toFind, arr[i]) == 0)
        .findFirst()
        .orElse(-1);
}
```

101. Проверка эквивалентности или несовпадения двух массивов

Два массива примитивов эквивалентны, если они содержат одинаковое число элементов и все соответствующие пары элементов в двух массивах эквивалентны.

Решения этих двух задач зависят от служебного класса `Arrays`. В следующих далее разделах приведены решения этих задач.

Проверка эквивалентности двух массивов

Проверка эквивалентности двух массивов может быть легко выполнена посредством метода `Arrays.equals()`. Этот флаговый метод идет во многих вариантах для примитивных типов, объектов и обобщений. Он также поддерживает компараторы.

Давайте рассмотрим следующие три массива целых чисел:

```
int[] integers1 = {3, 4, 5, 6, 1, 5};
int[] integers2 = {3, 4, 5, 6, 1, 5};
int[] integers3 = {3, 4, 5, 6, 1, 3};
```

Теперь проверим эквивалентность массивов `integers1` и `integers2` и эквивалентность массивов `integers1` и `integers3` между собой. Это очень просто:

```
boolean i12 = Arrays.equals(integers1, integers2);    // true
boolean i13 = Arrays.equals(integers1, integers3);    // false
```

Приведенные выше примеры проверяют эквивалентность двух массивов, но мы также можем проверить эквивалентность двух сегментов (или интервалов) массивов посредством булевого метода `equals(int[] a, int aFromIndex, int aToIndex, int[] b, int bFromIndex, int bToIndex)`. Таким образом, мы разграничиваем сегмент первого массива посредством интервала `[aFromIndex, aToIndex)` и сегмент второго массива с помощью интервала `[bFromIndex, bToIndex)`:

```
// true
boolean i13 = Arrays.equals(integers1, 1, 4, integers3, 1, 4);
```

Теперь допустим, что у нас есть три массива объектов класса `Melon`:

```
public class Melon {
    private final String type;
    private final int weight;

    public Melon(String type, int weight) {
        this.type = type;
        this.weight = weight;
    }

    // геттеры, equals() and hashCode() опущены для краткости
}

Melon[] melons1 = {
    new Melon("Horned", 1500), new Melon("Gac", 1000)
};

Melon[] melons2 = {
    new Melon("Horned", 1500), new Melon("Gac", 1000)
};

Melon[] melons3 = {
    new Melon("Hami", 1500), new Melon("Gac", 1000)
};
```

Два массива объектов считаются эквивалентными на основе контракта метода `equals()` или на основе указанного компаратора. Мы можем легко проверить `melons1` на эквивалентность с `melons2`, и `melons1` на эквивалентность с `melons3` следующим образом:

```
boolean m12 = Arrays.equals(melons1, melons2);    // true
boolean m13 = Arrays.equals(melons1, melons3);    // false
```


И в явном интервале следует использовать булев метод `equals(Object[] a, int aFromIndex, int aToIndex, Object[] b, int bFromIndex, int bToIndex)`:

```
boolean msl3 = Arrays.equals(melons1, 1, 2, melons3, 1, 2); // false
```

В отличие от этих примеров, которые опираются на имплементацию метода `Melon.equals()`, следующие два примера опираются на вот эти два компаратора:

```
Comparator<Melon> byType = Comparator.comparing(Melon::getType);
```

```
Comparator<Melon> byWeight = Comparator.comparing(Melon::getWeight);
```

Используя булев метод `equals(T[] a, T[] a2, Comparator<? super T> cmp)`, мы имеем следующее:

```
boolean mw13 = Arrays.equals(melons1, melons3, byWeight); // true
```

```
boolean mt13 = Arrays.equals(melons1, melons3, byType); // false
```

И в явном интервале, используя `Comparator`, `<T> boolean equals(T[] a, int aFromIndex, int aToIndex, T[] b, int bFromIndex, int bToIndex, Comparator<? super T> cmp)`, мы имеем следующее:

```
// true
```

```
boolean mrt13 = Arrays.equals(melons1, 1, 2, melons3, 1, 2, byType);
```

Проверка несовпадения в двух массивах

Если два массива эквивалентны, то несовпадение должно возвращать `-1`. Но если два массива не эквивалентны, то несовпадение должно возвращать индекс первого несовпадения между двумя заданными массивами. Для решения этой задачи мы можем опираться на методы `Arrays.mismatch()` JDK 9.

Например, мы можем проверить несовпадение массивов `integers1` и `integers2` следующим образом:

```
int ml12 = Arrays.mismatch(integers1, integers2); // -1
```

Результат равен `-1`, т. к. массивы `integers1` и `integers2` равны. Но если мы проверим `integers1` и `integers3`, то получим значение `5`, которое является индексом первого несовпадения этих двух массивов:

```
int ml13 = Arrays.mismatch(integers1, integers3); // 5
```



Если заданные массивы имеют разную длину и меньший является префиксом для большего, то возвращаемое несовпадение является длиной меньшего массива.

Для массивов объектов также существует специальная связка методов `mismatch()`. Эти методы рассчитывают на контракт метода `equals()` или на заданный компаратор. Мы можем проверить наличие несовпадения между `melons1` и `melons2` следующим образом:

```
int mm12 = Arrays.mismatch(melons1, melons2); // -1
```

Если несовпадение происходит на первом индексе, то возвращаемое значение равно `0`. Это происходит в случае `melons1` и `melons3`:

```
int mm13 = Arrays.mismatch(melons1, melons3); // 0
```

Как и в случае с методом `Arrays.equals()`, мы можем проверить наличие несовпадений в явном интервале с помощью компаратора:

```
// диапазон [1, 2), вернуть -1
int mms13 = Arrays.mismatch(melons1, 1, 2, melons3, 1, 2);

// Comparator по весу дыни, вернуть -1
int mmw13 = Arrays.mismatch(melons1, melons3, byWeight);

// Comparator по сортам дынь, вернуть 0
int mmt13 = Arrays.mismatch(melons1, melons3, byType);

// диапазон [1,2) и Comparator по сортам дынь, вернуть -1
int mmrt13 = Arrays.mismatch(melons1, 1, 2, melons3, 1, 2, byType);
```

102. Сравнение двух массивов лексикографически

Начиная с JDK 9, мы можем сравнивать два массива лексикографически посредством методов `Arrays.compare()`. Поскольку нет необходимости изобретать велосипед, просто обновитесь до JDK 9. И давайте углубимся в это сравнение.

Лексикографическое сравнение двух массивов может возвращать следующее:

- ◆ 0, если заданные массивы равны и содержат одинаковые элементы в одинаковом порядке;
- ◆ значение меньше 0, если первый массив лексикографически меньше второго массива;
- ◆ значение больше 0, если первый массив лексикографически больше второго массива.

Если длина первого массива меньше длины второго массива, то первый массив лексикографически меньше второго массива. Если массивы имеют одинаковую длину, содержат примитивы и имеют общий префикс, то лексикографическое сравнение является результатом сравнения двух элементов, а именно как `Integer.compare(int, int)`, `Boolean.compare(boolean, boolean)`, `Byte.compare(byte, byte)` и т. д. Если массивы содержат объекты, то лексикографическое сравнение опирается на заданный компаратор или на имплементацию интерфейса `Comparable`.

Для начала рассмотрим следующие массивы примитивов:

```
int[] integers1 = {3, 4, 5, 6, 1, 5};
int[] integers2 = {3, 4, 5, 6, 1, 5};
int[] integers3 = {3, 4, 5, 6, 1, 3};
```

Сейчас массив `integers1` лексикографически равен массиву `integers2`, потому что они эквивалентны и содержат одинаковые элементы в одном и том же порядке, `int compare(int[] a, int[] b)`:

```
int i12 = Arrays.compare(integers1, integers2); // 0
```

Однако массив `integers1` лексикографически больше массива `integers3`, т. к. они имеют одинаковый префикс (3, 4, 5, 6, 1), но для последнего элемента, метод `Integer.compare(5,3)` возвращает значение больше 0, поскольку 5 больше 3:

```
int i13 = Arrays.compare(integers1, integers3); // 1
```

Лексикографическое сравнение может выполняться на разных интервалах массивов. Например, в следующем ниже примере сравниваются `integers1` и `integers3` в интервале [3; 6) посредством метода `int compare(int[] a, int aFromIndex, int aToIndex, int[] b, int bFromIndex, int bToIndex)`:

```
int is13 = Arrays.compare(integers1, 3, 6, integers3, 3, 6); // 1
```

Для массивов объектов класс `Arrays` также предусматривает набор выделенных методов `compare()`. Помните класс `Melon`? Так вот, для того чтобы сравнить два массива объектов класса `Melon` без явного компаратора, нам нужно имплементировать интерфейс `Comparable` и метод `compareTo()`. Допустим, что мы опираемся на веса дынь следующим образом:

```
public class Melon implements Comparable {
    private final String type;
    private final int weight;

    @Override
    public int compareTo(Object o) {
        Melon m = (Melon) o;

        return Integer.compare(this.getWeight(), m.getWeight());
    }

    // конструктор, геттеры, equals() и hashCode () опущены для краткости
}
```



Обратите внимание, что лексикографическое сравнение массивов объектов не опирается на метод `equals()`. Оно требует явного компаратора или элементов `Comparable`.

Допустим, что у нас есть следующие ниже массивы объектов класса `Melon`:

```
Melon[] melons1 = {new Melon("Horned", 1500), new Melon("Gac", 1000)};
Melon[] melons2 = {new Melon("Horned", 1500), new Melon("Gac", 1000)};
Melon[] melons3 = {new Melon("Hami", 1600), new Melon("Gac", 800)};
```

Сравним лексикографически массив `melons1` с массивом `melons2` посредством метода `<T extends Comparable<? super T>> int compare(T[] a, T[] b)`:

```
int m12 = Arrays.compare(melons1, melons2); // 0
```

Поскольку массивы `melons1` и `melons2` являются идентичными, результат равен 0.

Теперь давайте сделаем то же самое с массивами `melons1` и `melons3`. На этот раз результат будет отрицательным, а это означает, что массив `melons1` лексикографи-

чески меньше массива `melons3`. Это верно, т. к. в индексе 0 дыня `Horned` имеет вес 1500 г, что меньше веса дыни `Nami`, равного 1600 г:

```
int m13 = Arrays.compare(melons1, melons3); // -1
```

Мы можем выполнить сравнение в разных интервалах массивов посредством метода `<T extends Comparable<? super T>> int compare(T[] a, int aFromIndex, int aToIndex, T[] b, int bFromIndex, int bToIndex)`. Например, в общем интервале `[1; 2]` `melons1` лексикографически больше `melons2`, т. к. вес `Gas` составляет 1000 г в `melons1` и 800 г в `melons3`:

```
int m13 = Arrays.compare(melons1, 1, 2, melons3, 1, 2); // 1
```

Если мы не хотим опираться на элементы `Comparable` (с имплементацией интерфейса `Comparable`), то можем передать компаратор посредством метода `<T> int compare(T[] a, T[] b, Comparator<? super T> cmp)`:

```
Comparator<Melon> byType = Comparator.comparing(Melon::getType);
int m13 = Arrays.compare(melons1, melons3, byType); // 14
```

Использовать интервалы также можно посредством метода `<T> int compare(T[] a, int aFromIndex, int aToIndex, T[] b, int bFromIndex, int bToIndex, Comparator<? super T> cmp)`:

```
int m13 = Arrays.compare(melons1, 1, 2, melons3, 1, 2, byType); // 0
```



Если массивы чисел должны трактоваться как беззнаковые, то используйте связку методов `Arrays.compareUnsigned()`, которые имеются для типов `byte`, `short`, `int` и `long`.

Для того чтобы сравнить две строки лексикографически, используйте методы `String.compareTo()` и `int compareTo(String anotherString)`.

103. Создание потока из массива

После того как мы создали объект `Stream` из массива, у нас будет доступ ко всем преимуществам API потоков. Следовательно, эту операцию удобно иметь под рукой, всегда готовой к применению в нашем арсенале инструментов.

Давайте начнем с массива строк (здесь могут быть и другие объекты):

```
String[] arr = {"One", "Two", "Three", "Four", "Five"};
```

Самый простой способ создать поток из этого массива `String[]` — опереться на метод `Arrays.stream()`, доступный, начиная с JDK 8:

```
Stream<String> stream = Arrays.stream(arr);
```

Как вариант, если нам нужен поток из подмассива, то просто следует добавить интервал в качестве аргументов. Например, давайте создадим поток из элементов, которые находятся в интервале между (0; 2) и равны "One" и "Two":

```
Stream<String> stream = Arrays.stream(arr, 0, 2);
```

Те же случаи, но с проходом через `List`, можно написать следующим образом:

```
Stream<String> stream = Arrays.asList(arr).stream();
Stream<String> stream = Arrays.asList(arr).subList(0, 2).stream();
```

Еще одно решение основано на методах `Stream.of()`, как в следующих ниже простых примерах:

```
Stream<String> stream = Stream.of(arr);
Stream<String> stream = Stream.of("One", "Two", "Three");
```

Создать массив из объекта `Stream` можно посредством метода `Stream.toArray()`. Например, простой подход выглядит следующим образом:

```
String[] array = stream.toArray(String[]::new);
```

В дополнение к этому давайте рассмотрим массив примитивов:

```
int[] integers = {2, 3, 4, 1};
```

В таком случае метод `Arrays.stream()` помогает снова, с той лишь разницей, что возвращаемый результат имеет тип `IntStream` (который является специализацией типа `Stream` для примитивного типа `int`):

```
IntStream intStream = Arrays.stream(integers);
```

Но класс `IntStream` также предоставляет метод `of()`, который можно использовать следующим образом:

```
IntStream intStream = IntStream.of(integers);
```

Иногда нам нужно определить поток последовательно упорядоченных целых чисел с шагом приращения, равным 1. Более того, размер потока должен быть равен размеру массива. Специально для таких случаев класс `IntStream` предусматривает два метода — `range(int inclusive, int exclusive)` и `rangeClosed(int startInclusive, int endInclusive)`:

```
IntStream intStream = IntStream.range(0, integers.length);
IntStream intStream = IntStream.rangeClosed(0, integers.length);
```

Создать массив из потока целых чисел можно посредством метода `Stream.toArray()`. Например, простой подход выглядит следующим образом:

```
int[] intArray = intStream.toArray();

// для упакованных целых чисел
int[] intArray = intStream.mapToInt(i -> i).toArray();
```



Помимо целочисленной специализации `IntStream` потока `Stream`, JDK 8 предусматривает специализации для типов `long` (`LongStream`) и `double` (`DoubleStream`).

104. Минимальное, максимальное и среднее значения массива

Операция вычисления минимального, максимального и среднего значений массива встречается очень часто. Рассмотрим несколько подходов к решению этой задачи в функциональном и императивном стилях программирования.

Вычисление максимума и минимума

Вычисление максимального значения массива чисел может быть имплементировано путем обхода массива в цикле и отслеживания максимального значения путем сравнения с каждым элементом массива. В терминах исходного кода это можно написать следующим образом:

```
public static int max(int[] arr) {
    int max = arr[0];

    for (int elem: arr) {
        if (elem > max) {
            max = elem;
        }
    }

    return max;
}
```

Чуть-чуть улучшим удобочитаемость и воспользуемся методом `Math.max()` вместо инструкции `if`:

```
...
max = Math.max(max, elem);
...
```

Предположим, что у нас есть следующий ниже массив целых чисел и служебный класс с именем `MathArrays`, который содержит приведенные выше методы:

```
int[] integers = {2, 3, 4, 1, -4, 6, 2};
```

Максимум этого массива можно легко получить следующим образом:

```
int maxInt = MathArrays.max(integers); // 6
```

В функциональном стиле Java 8 решение этой проблемы влечет за собой одну-единственную строку кода:

```
int maxInt = Arrays.stream(integers).max().getAsInt();
```



В подходе на основе функционального стиля метод `max()` возвращает `OptionalInt`. Точно так же у нас есть `OptionalLong` и `OptionalDouble`.

Далее давайте допустим, что у нас есть массив объектов, в данном случае массив объектов класса `Melon`:

```
Melon[] melons = {
    new Melon("Horned", 1500), new Melon("Gac", 2200),
    new Melon("Hami", 1600), new Melon("Gac", 2100)
};
```

```
public class Melon implements Comparable {
    private final String type;
    private final int weight;
```

```
@Override
public int compareTo(Object o) {
    Melon m = (Melon) o;
    return Integer.compare(this.getWeight(), m.getWeight());
}
```

```
// конструктор, геттеры, equals() и hashCode() опущены для краткости
}
```

Совершенно очевидно, что определенные нами ранее методы `max()` не могут быть использованы в данном случае, но логический принцип остается тем же самым. На этот раз имплементация должна опираться на `Comparable` или `Comparator`. Имплементация на основе `Comparable` может быть следующей:

```
public static <T extends Comparable<T>> T max(T[] arr) {
    T max = arr[0];

    for (T elem : arr) {
        if (elem.compareTo(max) > 0) {
            max = elem;
        }
    }
    return max;
}
```

Обратите внимание на метод `Melon.compareTo()` и на то, что наша имплементация будет сравнивать веса дынь. Поэтому мы можем легко найти самую тяжелую дыню из нашего массива следующим образом:

```
Melon maxMelon = MathArrays.max(melons); // Gac(2200g)
```

И имплементация, опирающаяся на компаратор, может быть написана следующим образом:

```
public static <T> T max(T[] arr, Comparator<? super T> c) {
    T max = arr[0];

    for (T elem: arr) {
        if (c.compare(elem, max) > 0) {
            max = elem;
        }
    }
    return max;
}
```

Если мы определим компаратор по сорту дынь, то получим следующее:

```
Comparator<Melon> byType = Comparator.comparing(Melon::getType);
```

Тогда мы получаем самую большую дыню, соответствующую лексикографическому сравнению строк:

```
Melon maxMelon = MathArrays.max(melons, byType); // Horned(1500g)
```

В функциональном стиле Java 8 решение этой задачи влечет за собой одну-единственную строку кода:

```
Melon maxMelon = Arrays.stream(melons).max(byType).orElseThrow();
```

Вычисление среднего значения

Вычисление среднего значения массива чисел (в данном случае целых чисел) может быть имплементировано в два простых шага:

1. Вычислить сумму элементов массива.
2. Разделить эту сумму на длину массива.

В исходном коде мы имеем следующее:

```
public static double average(int[] arr) {
    return sum(arr) / arr.length;
}
```

```
public static double sum(int[] arr) {
    double sum = 0;

    for (int elem: arr) {
        sum += elem;
    }
    return sum;
}
```

Среднее значение нашего массива целых чисел равно 2.0:

```
double avg = MathArrays.average(integers);
```

В функциональном стиле Java 8 решение этой задачи влечет за собой одну-единственную строку кода:

```
double avg = Arrays.stream(integers).average().getAsDouble();
```



Относительно поддержки сторонними библиотеками, пожалуйста, обратитесь к библиотеке Apache Common Lang и ее классу ArrayUtil и к библиотеке Guava Chars и ее классам Ints, Longs и другим классам.

105. Инвертирование массива

Существует несколько решений этой задачи. Некоторые из них мутируют исходный массив, тогда как другие просто возвращают новый массив.

Допустим, что у нас есть следующий ниже массив целых чисел:

```
int[] integers = {-1, 2, 3, 1, 4, 5, 3, 2, 22};
```

Начнем с простой имплементации, которая меняет местами первый элемент массива с последним элементом, второй элемент с предпоследним элементом и т. д.:

```
public static void reverse(int[] arr) {
    for (int leftHead = 0, rightHead = arr.length - 1;
        leftHead < rightHead; leftHead++, rightHead--) {
```



```
int elem = arr[leftHead];
arr[leftHead] = arr[rightHead];
arr[rightHead] = elem;
}
}
```

Приведенное выше решение мутирует заданный массив, и это поведение не всегда желательно. Разумеется, мы можем его модифицировать и вернуть новый массив, либо мы можем опереться на функциональный стиль Java 8 следующим образом:

```
// 22, 2, 3, 5, 4, 1, 3, 2, -1
int[] reversed = IntStream.rangeClosed(1, integers.length)
    .map(i -> integers[integers.length - i]).toArray();
```

Теперь давайте инвертируем массив объектов. Для этого рассмотрим класс `Melon`:

```
public class Melon {
    private final String type;
    private final int weight;

    // конструктор, геттеры, equals(), hashCode() опущены для краткости
}
```

Также давайте рассмотрим массив объектов класса `Melon`:

```
Melon[] melons = {
    new Melon("Crenshaw", 2000),
    new Melon("Gac", 1200),
    new Melon("Bitter", 2200)
};
```

Первый вариант решения этой задачи предполагает использование обобщений для формирования имплементации, которая меняет местами первый элемент массива с последним элементом, второй элемент с предпоследним элементом и т. д.:

```
public static <T> void reverse(T[] arr) {
    for (int leftHead = 0, rightHead = arr.length - 1;
        leftHead < rightHead; leftHead++, rightHead--) {

        T elem = arr[leftHead];
        arr[leftHead] = arr[rightHead];
        arr[rightHead] = elem;
    }
}
```

Поскольку наш массив содержит объекты, мы можем опереться и на метод `Collections.reverse()`. Нам просто нужно конвертировать массив в `List` посредством метода `Arrays.asList()`:

```
// Bitter(2200g), Gac(1200g), Crenshaw(2000g)
Collections.reverse(Arrays.asList(melons));
```

Два приведенных выше варианта решения мутируют элементы массива. Функциональный стиль Java 8 помогает нам избежать этой мутации:

```
// Bitter(2200g), Gac(1200g), Crenshaw(2000g)
Melon[] reversed = IntStream.rangeClosed(1, melons.length)
    .mapToObj(i -> melons[melons.length - i])
    .toArray(Melon[]::new);
```



Относительно поддержки сторонними библиотеками, пожалуйста, обратитесь к библиотеке `Apache Common Lang` и ее методу `ArrayUtils.reverse()`, а также к библиотеке `Guava` и ее классу `Lists`.

106. Заполнение и настройка массива

Иногда нам нужно заполнить массив фиксированным значением. Например, мы можем захотеть заполнить массив целых чисел значением 1. Самый простой способ выполнения этой задачи опирается на инструкцию `for` следующим образом:

```
int[] arr = new int[10];

// 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
for (int i = 0; i < arr.length; i++) {
    arr[i] = 1;
}
```

Но мы можем свести этот код к одной строке кода посредством связки методов `Arrays.fill()`. Указанный метод идет в разновидностях для примитивов и для объектов. Приведенный выше код можно переписать с использованием метода `Arrays.fill(int[] a, int val)` следующим образом:

```
// 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
Arrays.fill(arr, 1);
```



Метод `Arrays.fill()` также идет в разновидностях для заполнения только сегмента/интервала массива. Для целых чисел этот метод имеет сигнатуру `fill(int[] a, int fromIndexInclusive, int toIndexExclusive, int val)`.

Теперь как насчет применения генераторной функции для вычисления каждого элемента массива? Например, допустим, что мы хотим вычислить каждый элемент как предыдущий элемент плюс 1. Самый простой подход будет опираться на инструкцию `for` следующим образом:

```
// 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
for (int i = 1; i < arr.length; i++) {
    arr[i] = arr[i - 1] + 1;
}
```

Приведенный выше код придется модифицировать соответствующим образом в зависимости от вычислений, которые должны быть применены к каждому элементу.

Для таких операций JDK 8 идет со связкой методов `Arrays.setAll()` и `Arrays.parallelSetAll()`. Например, указанный выше фрагмент кода может быть переписан посредством метода `setAll(int[] array, IntUnaryOperator generator)` следующим образом:

```
// 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Arrays.setAll(arr, t -> {
    if (t == 0) {
        return arr[t];
    } else {
        return arr[t - 1] + 1;
    }
});
```



Помимо этого метода, у нас также есть методы `setAll(double[] array, IntToDoubleFunction generator)`, `setAll(long[] array, IntToLongFunction generator)` и `setAll(T[] array, IntFunction<? extends T> generator)`.

В зависимости от генераторной функции эта операция может выполняться параллельно с другим кодом. Например, приведенную выше генераторную функцию не получится применить параллельно, т. к. каждый элемент зависит от значения предыдущего элемента. Попытка применить эту генераторную функцию в параллельных расчетах приведет к неправильным и нестабильным результатам.

Но давайте допустим, что мы хотим взять указанный ранее массив (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) и умножить каждое четное значение на себя и уменьшить каждое нечетное значение на 1. Поскольку каждый элемент может быть вычислен индивидуально, в данном случае мы можем задействовать возможности параллельного процесса. Эта работа идеально подходит для методов `Arrays.parallelSetAll()`. В сущности, эти методы предназначены для параллелизации методов `Arrays.setAll()`.

Теперь применим `parallelSetAll(int[] array, IntUnaryOperator generator)` к этому массиву:

```
// 0, 4, 2, 16, 4, 36, 6, 64, 8, 100
Arrays.parallelSetAll(arr, t -> {
    if (arr[t] % 2 == 0) {
        return arr[t] * arr[t];
    } else {
        return arr[t] - 1;
    }
});
```



Для каждого метода `Arrays.setAll()` существует метод `Arrays.parallelSetAll()`.

В качестве бонуса массивы идут со связкой методов `parallelPrefix()`. Эти методы полезны для применения математической функции к элементам массива как кумулятивно, так и конкурентно.

Например, если мы хотим вычислить каждый элемент массива как сумму предыдущих элементов, мы можем сделать это следующим образом:

```
// 0, 4, 6, 22, 26, 62, 68, 132, 140, 240
Arrays.parallelPrefix(arr, (t, q) -> t + q);
```

107. Следующий больший элемент

Задача отыскания следующего большего элемента (next greater element, NGE) является классической задачей с привлечением массивов.

В принципе, имея массив и элемент *e* из него, мы хотим извлечь следующий (правый) элемент, больший, чем *e*. Например, допустим, что у нас есть такой массив:

```
int[] integers = {1, 2, 3, 4, 12, 2, 1, 4};
```

Выборка следующего большего элемента для каждого элемента приведет к парам (−1 интерпретируется как "ни один элемент с правой стороны не больше текущего"):

```
1 : 2 2 : 3 3 : 4 4 : 12 12 : -1 2 : 4 1 : 4 4 : -1
```

Простое решение этой задачи будет перебирать элементы массива в цикле до тех пор, пока не отыщется больший элемент либо пока не исчерпаются элементы для проверки. Если мы хотим просто печатать пары на экране, то можем написать тривиальный код такой, как показан ниже:

```
public static void println(int[] arr) {
    int nge;
    int n = arr.length;

    for (int i = 0; i < n; i++) {
        nge = -1;

        for (int j = i + 1; j < n; j++) {
            if (arr[i] < arr[j]) {
                nge = arr[j];
                break;
            }
        }

        System.out.println(arr[i] + " : " + nge);
    }
}
```

Еще один вариант решения опирается на стек. Мы помещаем элементы в стек до тех пор, пока обрабатываемый в данный момент элемент не станет больше верхнего элемента в стеке. Когда это происходит, мы выталкиваем этот элемент. Данный вариант решения имеется в исходном коде, прилагаемом к этой книге.

108. Изменение размера массива

Задачу увеличения размера массива нельзя назвать прямолинейной. Это обусловлено тем, что массивы Java имеют фиксированный размер и мы не можем его модифицировать. Решение этой задачи заключается в создании нового массива требуемого размера и копировании всех значений из исходного массива в новый. Это можно сделать посредством метода `Arrays.copyOf()` либо `System.arraycopy()` (используемого внутри метода `Arrays.copyOf()`).

Для массива примитивов (например, типа `int`) мы можем добавить значение в массив после увеличения его размера на 1 следующим образом:

```
public static int[] add(int[] arr, int item) {
    int[] newArr = Arrays.copyOf(arr, arr.length + 1);
    newArr[newArr.length - 1] = item;

    return newArr;
}
```

Как вариант, мы можем удалить последнее значение следующим образом:

```
public static int[] remove(int[] arr) {
    int[] newArr = Arrays.copyOf(arr, arr.length - 1);

    return newArr;
}
```

Кроме того, мы можем изменить размер массива на заданную длину следующим образом:

```
public static int[] resize(int[] arr, int length) {
    int[] newArr = Arrays.copyOf(arr, arr.length + length);

    return newArr;
}
```

Исходный код, прилагаемый к этой книге, также содержит альтернативы методу `System.arraycopy()`. Кроме того, в нем имеются имплементации обобщенных массивов. Их сигнатуры выглядят следующим образом:

```
public static <T> T[] addObject(T[] arr, T item);
public static <T> T[] removeObject(T[] arr);
public static <T> T[] resize(T[] arr, int length);
```

Пока мы находимся в благоприятном контексте, давайте обсудим смежную тему: как создавать обобщенный массив в Java. Приведенная далее инструкция работать не будет:

```
T[] arr = new T[arr_size]; // вызывает ошибку создания обобщенного массива
```

Существует несколько подходов, но Java использует следующий ниже код в методе `copyOf(T[] original, int newLength)`:

```
// newType равен original.getClass()
T[] copy = ((Object) newType == (Object) Object[].class) ?
    (T[]) new Object[newLength] :
    (T[]) Array.newInstance(newType.getComponentType(), newLength);
```

109. Создание немодифицируемых/немутируемых коллекций

Немодифицируемые/немутируемые коллекции в Java легко создаются посредством методов `Collections.unmodifiableFoo()` (например, `unmodifiableList()`) и, начиная с JDK 9, посредством набора методов `of()` из интерфейсов `List`, `Set`, `Map` и других.

Далее мы будем использовать эти методы в ряде примеров для получения немодифицируемых/немутируемых коллекций. Главная цель состоит в том, чтобы выяснить, является ли каждая определенная коллекция немодифицируемой или немутуруемой.



Перед чтением этого раздела рекомендуется ознакомиться с задачами, посвященными немутуруемости, из главы 2.

ОК. В случае с примитивами все довольно просто. Например, мы можем создать немутуруемый список целых чисел так:

```
private static final List<Integer> LIST
    = Collections.unmodifiableList(Arrays.asList(1, 2, 3, 4, 5));

private static final List<Integer> LIST = List.of(1, 2, 3, 4, 5);
```

Для следующих далее примеров рассмотрим вот такой мутируемый класс:

```
public class MutableMelon {
    private String type;
    private int weight;

    // конструктор опущен для краткости
    public void setType(String type) {
        this.type = type;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }

    // геттеры, equals() и hashCode() опущены для краткости
}
```

Задача 1 (*`Collections.unmodifiableList()`*)

Давайте создадим список объектов `MutableMelon` посредством метода `Collections.unmodifiableList()`:

```
// Crenshaw(2000g), Gac(1200g)
private final MutableMelon melon1 = new MutableMelon("Crenshaw", 2000);
private final MutableMelon melon2 = new MutableMelon("Gac", 1200);

private final List<MutableMelon> list
    = Collections.unmodifiableList(Arrays.asList(melon1, melon2));
```

Итак, каким является этот список: немодифицируемым или немутлируемым? Ответ — он является немодифицируемым. В отличие от методов записи, которые будут выбрасывать исключение `UnsupportedOperationException`, приведенные ниже объекты `melon1` и `melon2` можно мутлировать. Например, давайте установим вес наших дынь равным 0:

```
melon1.setWeight(0);  
melon2.setWeight(0);
```

Теперь список покажет вот такие дыни (таким образом, этот список был мутлирован):

```
Crenshaw(0g), Gac(0g)
```

Задача 2 (`Arrays.asList()`)

Давайте создадим список объектов `MutableMelon`, жестко закодировав экземпляры непосредственно в `Arrays.asList()`:

```
private final List<MutableMelon> list  
    = Collections.unmodifiableList(Arrays.asList(  
        new MutableMelon("Crenshaw", 2000),  
        new MutableMelon("Gac", 1200)));
```

Итак, каким является этот список: немодифицируемым или немутлируемым? Ответ — он является немодифицируемым. В отличие от методов записи, которые будут выбрасывать исключение `UnsupportedOperationException`, жестко закодированные экземпляры могут быть доступны посредством метода `List.get()`. После того как к ним будет получен доступ, они могут быть мутлированы:

```
MutableMelon melon1 = list.get(0);  
MutableMelon melon2 = list.get(1);
```

```
melon1.setWeight(0);  
melon2.setWeight(0);
```

Теперь этот список покажет вот такие дыни (таким образом, этот список был мутлирован):

```
Crenshaw(0g), Gac(0g)
```

Задача 3 (`Collections.unmodifiableList()` и статический блок)

Давайте создадим список объектов `MutableMelon` посредством метода `Collections.unmodifiableList()` и статического блока:

```
private static final List<MutableMelon> list;  
static {  
    final MutableMelon melon1 = new MutableMelon("Crenshaw", 2000);  
    final MutableMelon melon2 = new MutableMelon("Gac", 1200);  
    list = Collections.unmodifiableList(Arrays.asList(melon1, melon2));  
}
```

Итак, каким является этот список: немодифицируемым или немутуируемым? Ответ — он является немодифицируемым. В отличие от методов записи, которые будут выбрасывать исключение `UnsupportedOperationException`, жестко закодированные экземпляры по-прежнему могут быть доступны посредством метода `List.get()`. После того как к ним будет получен доступ, они могут быть мутированы:

```
MutableMelon melon1l = list.get(0);
MutableMelon melon2l = list.get(1);
```

```
melon1l.setWeight(0);
melon2l.setWeight(0);
```

Теперь этот список покажет вот такие дыни (таким образом, этот список был мутирован):

```
Crenshaw(0g), Gac(0g)
```

Задача 4 (`List.of()`)

Давайте создадим список объектов `MutableMelon` посредством метода `List.of()`:

```
private final MutableMelon melon1 = new MutableMelon("Crenshaw", 2000);
private final MutableMelon melon2 = new MutableMelon("Gac", 1200);
```

```
private final List<MutableMelon> list = List.of(melon1, melon2);
```

Итак, каким является этот список: немодифицируемым или немутуируемым? Ответ — он является немодифицируемым. В отличие от методов записи, которые будут выбрасывать исключение `UnsupportedOperationException`, жестко закодированные экземпляры по-прежнему могут быть доступны посредством метода `List.get()`. После того как к ним будет получен доступ, они могут быть мутированы:

```
MutableMelon melon1l = list.get(0);
MutableMelon melon2l = list.get(1);
```

```
melon1l.setWeight(0);
melon2l.setWeight(0);
```

Теперь этот список покажет вот такие дыни (таким образом, этот список был мутирован):

```
Crenshaw(0g), Gac(0g)
```

* * *

Для следующих далее примеров рассмотрим вот такой немутуируемый класс:

```
public final class ImmutableMelon {
    private final String type;
    private final int weight;

    // конструктор, геттеры, equals() и hashCode() опущены для краткости
}
```


Задача 5 (немутируемое)

Давайте теперь создадим список немутуируемых объектов посредством методов `Collections.unmodifiableList()` и `List.of()`:

```
private static final ImmutableMelon MELON_1
    = new ImmutableMelon("Crenshaw", 2000);
private static final ImmutableMelon MELON_2
    = new ImmutableMelon("Gac", 1200);

private static final List<ImmutableMelon> LIST
    = Collections.unmodifiableList(Arrays.asList(MELON_1, MELON_2));
private static final List<ImmutableMelon> LIST
    = List.of(MELON_1, MELON_2);
```

Итак, каким является этот список: немодифицируемым или немутуируемым? Ответ — он является немодифицируемым. Методы записи будут выбрасывать исключение `UnsupportedOperationException`, и мы не можем мутировать экземпляры класса `ImmutableMelon`.



TIP

В качестве общего правила следует запомнить, что коллекция является немодифицируемой, если она определена с помощью методов `unmodifiableFoo()` или `of()` и содержит мутируемые данные, а также она является немутуируемой, если она немодифицируема и содержит немутуируемые данные (включая примитивы).

Обратите внимание на то, что непроницаемая немутуируемость должна учитывать API рефлексии Java и похожие API, которые имеют дополнительные полномочия в манипулировании кодом.

Относительно поддержки сторонними библиотеками, пожалуйста, обратитесь к библиотеке `Apache Common Collection` и ее классу `UnmodifiableList` (и его сопутствующим методам), а также к библиотеке `Guava` и ее классу `ImmutableList` (и его сопутствующим методам).

В случае отображения `Map` мы можем создать немодифицируемое/немутуируемое отображение `Map` посредством методов `unmodifiableMap()` или `Map.of()`.

Но мы также можем создать немутуируемое пустое отображение `Map` посредством метода `Collections.emptyMap()`:

```
Map<Integer, MutableMelon> emptyMap = Collections.emptyMap();
```



Подобно методу `emptyMap()`, мы имеем методы `Collections.emptyList()` и `Collections.emptySet()`. Эти методы очень удобны в качестве возвращаемых в методах, которые возвращают коллекции `Map`, `List` или `Set`, и когда мы хотим избежать возврата `null`.

В качестве альтернативы мы можем создать немодифицируемое/немутуируемое отображение `Map` с одним элементом посредством метода `Collections.singletonMap(K key, V value)`:

```
// немодифицируемая
Map<Integer, MutableMelon> mapOfSingleMelon
    = Collections.singletonMap(1, new MutableMelon("Gac", 1200));
```

```
// немутируемая
```

```
Map<Integer, ImmutableMelon> mapOfSingleMelon
    = Collections.singletonMap(1, new ImmutableMelon("Gac", 1200));
```



Аналогично методу `singletonMap()`, существуют методы `singletonList()` и `singleton()`. Последний предназначен для коллекции `Set`.

Кроме того, начиная с JDK 9, мы можем создавать немодифицируемое отображение `Map` посредством метода с именем `ofEntries()`. Этот метод берет элемент `Map.Entry` в качестве аргумента, как показано в следующем примере:

```
// unmodifiable Map.Entry containing the given key and value
import static java.util.Map.entry;

...
```

```
Map<Integer, MutableMelon> mapOfMelon = Map.ofEntries(
    entry(1, new MutableMelon("Apollo", 3000)),
    entry(2, new MutableMelon("Jade Dew", 3500)),
    entry(3, new MutableMelon("Cantaloupe", 1500))
);
```

В качестве альтернативы, немутуируемое отображение `Map` является еще одним вариантом:

```
Map<Integer, ImmutableMelon> mapOfMelon = Map.ofEntries(
    entry(1, new ImmutableMelon("Apollo", 3000)),
    entry(2, new ImmutableMelon("Jade Dew", 3500)),
    entry(3, new ImmutableMelon("Cantaloupe", 1500))
);
```

В дополнение к этому немодифицируемое/немутуируемое отображение `Map` может быть получено из модифицируемого/мутаемого отображения `Map` посредством метода `Map.copyOf(Map<? extends K, ? extends V> map)` из JDK 10:

```
Map<Integer, ImmutableMelon> mapOfMelon = new HashMap<>();
mapOfMelon.put(1, new ImmutableMelon("Apollo", 3000));
mapOfMelon.put(2, new ImmutableMelon("Jade Dew", 3500));
mapOfMelon.put(3, new ImmutableMelon("Cantaloupe", 1500));
```

```
Map<Integer, ImmutableMelon> immutableMapOfMelon
    = Map.copyOf(mapOfMelon);
```

В качестве бонуса в этом разделе давайте поговорим о немутуируемом массиве.

Вопрос: можно ли создать немутуируемый массив в Java?

Ответ: нет, нельзя. Или... один способ сделать немутуируемый массив в Java все-таки есть:

```
static final String[] immutable = new String[0];
```

Таким образом, все полезные массивы в Java являются мутаруемыми. Но мы можем сформировать вспомогательный класс для создания немутуируемых массивов,

основываясь на методе `Arrays.copyOf()`, который копирует элементы и создает новый массив (за кулисами этот метод опирается на метод `System.arraycopy()`).

Наш вспомогательный класс выглядит следующим образом:

```
import java.util.Arrays;

public final class ImmutableArray<T> {
    private final T[] array;

    private ImmutableArray(T[] a) {
        array = Arrays.copyOf(a, a.length);
    }

    public static <T> ImmutableArray<T> from(T[] a) {
        return new ImmutableArray<>(a);
    }

    public T get(int index) {
        return array[index];
    }

    // equals(), hashCode() и toString() опущены для краткости
}
```

Пример его использования выглядит следующим образом:

```
ImmutableArray<String> sample =
    ImmutableArray.from(new String[] {
        "a", "b", "c"
    });
```

110. Возврат значения по умолчанию из коллекции *Map*

До JDK 8 решение этой задачи опиралось на вспомогательный метод, который в сущности проверяет наличие заданного ключа в отображении `Map` и возвращает соответствующее значение либо значение по умолчанию. Такой метод может быть написан в служебном классе либо путем расширения интерфейса `Map`. Возвращая значение по умолчанию, мы избегаем возврата `null`, если заданный ключ не найден в отображении `Map`. Более того, этот подход удобен тем, что позволяет опираться на значения по умолчанию или конфигурацию.

Начиная с JDK 8, решение этой задачи состоит из простого вызова метода `Map.getOrDefault()`. Этот метод получает два аргумента, представляющих ключ для поиска в отображении `Map` и значение по умолчанию. Дефолтное значение действует как резервное значение, которое должно быть возвращено, когда заданный ключ не найден.

Например, допустим, что следующее ниже отображение `Map` обертывает несколько баз данных и их `host:port`, установленный по умолчанию:

```
Map<String, String> map = new HashMap<>();
map.put("postgresql", "127.0.0.1:5432");
map.put("mysql", "192.168.0.50:3306");
map.put("cassandra", "192.168.1.5:9042");
```

И давайте попробуем посмотреть, не содержит ли это отображение `Map` и `host:port` по умолчанию для базы данных `Derby`:

```
map.get("derby"); // null
```

Поскольку база данных `Derby` в коллекции `Map` отсутствует, результат будет равен `null`. Это не тот результат, который нам нужен. На самом деле, когда искомая база данных в отображении `Map` отсутствует, мы можем использовать `MongoDB` по адресу `69:89.31.226:27017`, который всегда доступен. Теперь мы можем легко сформировать это поведение следующим образом:

```
// 69:89.31.226:27017
String hpl = map.getOrDefault("derby", "69:89.31.226:27017");

// 192.168.0.50:3306
String hp2 = map.getOrDefault("mysql", "69:89.31.226:27017");
```



Этот метод удобен для построения гибких выражений и позволяет избежать нарушения работы кода при проверке на `null`. Обратите внимание, что возврат значения по умолчанию не равносильно тому, что это значение будет добавлено в отображение `Map`. Отображение `Map` остается немодифицированным.

111. Вычисление отсутствия/присутствия элемента в отображении *Map*

Иногда отображение `Map` не содержит точного готового элемента, который нам нужен. Более того, когда элемент отсутствует, возврат элемента по умолчанию также не является подходящим вариантом. Есть случаи, когда наш элемент требуется вычислить.

Для таких случаев `JDK 8` идет в комплекте со связкой методов `compute()`, `computeIfAbsent()`, `computeIfPresent()` и `merge()`. Правильный выбор между этими методами — это вопрос знания каждого из них.

Давайте теперь взглянем на имплементацию этих методов на примерах.

Пример 1 (`computeIfPresent()`)

Предположим, что у нас есть следующее ниже отображение `Map`:

```
Map<String, String> map = new HashMap<>();
map.put("postgresql", "127.0.0.1");
map.put("mysql", "192.168.0.50");
```

Мы используем это отображение для построения URL-адресов JDBC для различных баз данных.

Допустим, что мы хотим построить URL-адрес JDBC для базы данных MySQL. Если ключ `mysql` в этом отображении присутствует, то URL-адрес JDBC должен быть вычислен на основе соответствующего значения, `jdbc:mysql://192.168.0.50/customers_db`. Но если ключ `mysql` отсутствует, то URL-адрес JDBC должен быть равен `null`. Кроме того, если результат нашего вычисления равен `null` (URL-адрес JDBC вычислить не получается), то мы хотим удалить этот элемент из отображения.

Эта работа подходит для метода `V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)`.

В нашем случае бифункция `BiFunction`, используемая для вычисления нового значения, будет выглядеть следующим образом (`k` — это ключ из отображения, `v` — значение, ассоциированное с ключом):

```
BiFunction<String, String, String> jdbcUrl
    = (k, v) -> "jdbc:" + k + "://" + v + "/customers_db";
```

После того как мы добавим эту функцию, мы сможем вычислить новое значение для ключа `mysql` следующим образом:

```
// jdbc:mysql://192.168.0.50/customers_db
String mySqlJdbcUrl = map.computeIfPresent("mysql", jdbcUrl);
```

Поскольку ключ `mysql` в отображении присутствует, результатом будет `jdbc:mysql://192.168.0.50/customers_db`, и новое отображение содержит следующие ниже элементы:

```
postgresql=127.0.0.1, mysql=jdbc:mysql://192.168.0.50/customers_db
```



Вызов метода `computeIfPresent()` снова вычислит значение, а значит, в результате будет получено что-то вроде `mysql=jdbc:mysql://jdbc:mysql://...`. Очевидно, что этот результат не является нормальным, поэтому данный аспект следует учитывать.

С другой стороны, если мы применим то же самое вычисление для элемента, который не существует (например, `voltdb`), то будет возвращено значение `null`, и отображение останется нетронутым:

```
// null
String voltdbJdbcUrl = map.computeIfPresent("voltdb", jdbcUrl);
```

Пример 2 (`computeIfAbsent()`)

Предположим, что у нас есть следующее отображение `Map`:

```
Map<String, String> map = new HashMap<>();
map.put("postgresql", "jdbc:postgresql://127.0.0.1/customers_db");
map.put("mysql", "jdbc:mysql://192.168.0.50/customers_db");
```

Мы используем это отображение для построения URL-адресов JDBC для различных баз данных.

Допустим, что мы хотим построить URL-адрес JDBC для базы данных MongoDB. На этот раз, если ключ `mongodb` в отображении присутствует, то соответствующее значение должно быть возвращено без дальнейших вычислений. Но если этот ключ отсутствует (либо ассоциирован со значением `null`), то значение должно быть вычислено на основе этого ключа и текущего IP и добавлено в отображение. Если вычисленное значение равно `null`, то значение `null` является возвращаемым результатом и отображение остается нетронутым.

Что ж, эта работа подходит для метода `V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)`.

В нашем случае `Function`, используемая для вычисления значения, будет выглядеть следующим образом (первое значение типа `String` — это ключ из отображения (`k`), а второе значение типа `String` — значение, вычисляемое для этого ключа):

```
String address = InetAddress.getLocalHost().getHostAddress();
```

```
Function<String, String> jdbcUrl
    = k -> k + "://" + address + "/customers_db";
```

Основываясь на этой функции, мы можем попытаться получить URL-адрес JDBC для MongoDB посредством ключа `mongodb` следующим образом:

```
// mongodb://192.168.100.10/customers_db
String mongodbJdbcUrl = map.computeIfAbsent("mongodb", jdbcUrl);
```

Поскольку наше отображение `Map` не содержит ключа `mongodb`, он будет вычислен и добавлен в это отображение.

Если наша функция вычисляет значение `null`, то отображение остается нетронутым и возвращается значение `null`.



TIP

Повторный вызов функции `computeIfAbsent()` не будет вычислять значения повторно. Поскольку на этот раз `mongodb` находится в отображении `Map` (это значение было добавлено предыдущим вызовом), возвращается значение `MongoDB://192.168.100.10/customers_db`. Оно является одинаковым тому, которое получается при попытке извлечь URL-адрес JDBC для `mysql`, которая вернет `jdbc:mysql://192.168.0.50/customers_db` без дальнейших вычислений.

Пример 3 (`compute()`)

Предположим, что у нас есть следующее отображение `Map`:

```
Map<String, String> map = new HashMap<>();
map.put("postgresql", "127.0.0.1");
map.put("mysql", "192.168.0.50");
```

Мы используем это отображение для построения URL-адресов JDBC для различных баз данных.

Допустим, что мы хотим построить URL-адреса JDBC для баз данных MySQL и Derby. В данном случае независимо от того, присутствует ли ключ (`mysql` или `derby`) в отображении, URL-адрес JDBC должен быть вычислен на основе соответствующих ключа и значения (которое может быть равно `null`). В дополнение к этому, если ключ в отображении присутствует и результат нашего вычисления равен `null` (URL-адрес JDBC вычислить не получается), то мы хотим удалить этот элемент из отображения. В сущности, это представляет собой комбинацию методов `computeIfPresent()` и `computeIfAbsent()`.

Эта работа подходит для метода `V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)`.

На этот раз функция `BiFunction` должна быть написана так, чтобы она охватывала случай, когда значение искомого ключа равно `null`:

```
String address = InetAddress.getLocalHost().getHostAddress();
BiFunction<String, String, String> jdbcUrl = (k, v)
    -> "jdbc:" + k + "://" + ((v == null) ? address : v)
        + "/customers_db";
```

Теперь давайте вычислим URL-адрес JDBC для базы данных MySQL. Поскольку ключ `mysql` в отображении присутствует, вычисление будет опираться на соответствующее значение, например `192.168.0.50`. В результате будет обновлено значение ключа `mysql` в отображении:

```
// jdbc:mysql://192.168.0.50/customers_db
String mysqlJdbcUrl = map.compute("mysql", jdbcUrl);
```

В дополнение к этому давайте вычислим URL-адрес JDBC для базы данных Derby. Поскольку ключ `derby` в отображении отсутствует, вычисление будет опираться на текущий IP. Результат будет добавлен в отображение под ключом `derby`:

```
// jdbc:derby://192.168.100.10/customers_db
String derbyJdbcUrl = map.compute("derby", jdbcUrl);
```

После этих двух вычислений отображение будет содержать следующие три элемента:

- ◆ `postgresql=127.0.0.1;`
- ◆ `derby=jdbc:derby://192.168.100.10/customers_db;`
- ◆ `mysql=jdbc:mysql://192.168.0.50/customers_db.`



Обратите внимание на то, что вызов функции `compute()` снова приведет к пересчету значений. Это может дать нежелательные результаты, такие как `jdbc:derby://jdbc:derby://...`. Если результат вычисления равен `null` (например, URL-адрес JDBC вычислить не получается) и ключ (например, `mysql`) в отображении существует, то этот элемент будет удален из отображения `Map`, и возвращаемый результат будет равен `null`.

Пример 4 (*merge()*)

Предположим, что у нас есть следующее отображение Map:

```
Map<String, String> map = new HashMap<>();
map.put("postgresql", "9.6.1 ");
map.put("mysql", "5.1 5.2 5.6 ");
```

Мы используем это отображение для хранения версий каждого вида базы данных, разделенных пробелом.

Теперь допустим, что всякий раз, когда выпускается новая версия базы данных, мы хотим добавить ее в наше отображение Map под соответствующим ключом. Если ключ (например, `mysql`) в отображении присутствует, то мы хотим просто конкатенировать новую версию в конец текущего значения. Если ключ (например, `derby`) в отображении отсутствует, то в этом случае мы хотим просто его добавить.

Эта работа идеально подходит для метода `V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)`.

Если заданный ключ (`k`) не ассоциирован со значением либо ассоциирован с `null`, то новое значение будет равно `v`. Если заданный ключ (`k`) ассоциирован с непустым значением (не-`null`), то новое значение вычисляется на основе заданной бифункции `BiFunction`. Если результат этой функции равен `null`, и ключ в отображении присутствует, то этот элемент будет из отображения удален.

В нашем случае мы хотим конкатенировать текущее значение с новой версией, поэтому нашу бифункцию `BiFunction` можно написать следующим образом:

```
BiFunction<String, String, String> jdbcUrl = String::concat;
```

У нас похожая ситуация со следующим:

```
BiFunction<String, String, String> jdbcUrl
    = (vold, vnew) -> vold.concat(vnew);
```

Например, предположим, что мы хотим конкатенировать в отображении Map версию 8.0 базы данных MySQL. Это можно выполнить так:

```
// 5.1 5.2 5.6 8.0
String mysqlVersion = map.merge("mysql", "8.0 ", jdbcUrl);
```

Позже мы также конкатенируем версию 9.0:

```
// 5.1 5.2 5.6 8.0 9.0
String mysqlVersion = map.merge("mysql", "9.0 ", jdbcUrl);
```

Как вариант, мы добавляем версию 10.11.1.1 базы данных Derby. Это приведет к новому элементу в отображении, т. к. ключ `derby` не присутствует:

```
// 10.11.1.1
String derbyVersion = map.merge("derby", "10.11.1.1 ", jdbcUrl);
```

В конце этих трех операций элементы отображения будут выглядеть следующим образом:

```
postgresql=9.6.1, derby=10.11.1.1, mysql=5.1 5.2 5.6 8.0 9.0
```


Пример 5 (*putIfAbsent()*)

Предположим, что у нас есть следующее отображение Map:

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "postgresql");
map.put(2, "mysql");
map.put(3, null);
```

Мы используем это отображение для хранения имен некоторых разновидностей баз данных.

Теперь предположим, что мы хотим включить в это отображение больше разновидностей баз данных, основываясь на следующих ограничениях:

- ◆ если заданный ключ в отображении присутствует, то просто вернуть соответствующее значение и оставить отображение нетронутым;
- ◆ если заданный ключ в отображении отсутствует (либо ассоциирован со значением null), то поместить заданное значение в отображение и вернуть null.

Что ж, эта работа подходит для метода `putIfAbsent(K key, V value)`.

Следующие ниже три попытки говорят сами за себя:

```
String v1 = map.putIfAbsent(1, "derby");      // postgresql
String v2 = map.putIfAbsent(3, "derby");      // null
String v3 = map.putIfAbsent(4, "cassandra");  // null
```

А содержимое отображения Map выглядит следующим образом:

```
1=postgresql, 2=mysql, 3=derby, 4=cassandra
```

112. Удаление элемента из отображения Map

Удаление элемента из отображения Map выполняется по ключу либо по ключу и значению.

Например, допустим, что у нас есть следующее отображение Map:

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "postgresql");
map.put(2, "mysql");
map.put(3, "derby");
```

Удаление по ключу выполняется так же просто, как вызов метода `V Map.remove(Object key)`. Если элемент, соответствующий заданному ключу, успешно удален, то этот метод возвращает ассоциированное значение, в противном случае он возвращает null.

Взгляните на примеры:

```
String r1 = map.remove(1); // postgresql
String r2 = map.remove(4); // null
```

Теперь отображение `Map` содержит следующие элементы (данные по ключу `1` были удалены):

```
2=mysql, 3=derby
```

Начиная с JDK 8, интерфейс `Map` был обогащен новым флаговым методом `remove()` с сигнатурой `boolean remove(Object key, Object value)`. Используя этот метод, мы можем удалить элемент из отображения только в том случае, если существует идеальное совпадение между заданным ключом и значением. Этот метод представляет собой укороченную форму следующих условий: `map.containsKey(key) && Objects.equals(map.get(key), value)`.

Приведем два простых примера:

```
// true
boolean r1 = map.remove(2, "mysql");

// false (ключ присутствует, но значения не совпадают)
boolean r2 = map.remove(3, "mysql");
```

Результирующее отображение `Map` содержит единственный оставшийся элемент, `3=derby`.

Обход в цикле и удаление из отображения `Map` можно выполнить по крайней мере двумя способами: во-первых, посредством итератора (программное решение присутствует в прилагаемом исходном коде), а во-вторых, начиная с JDK 8, мы можем сделать это посредством метода `removeIf(Predicate<? super E> filter)`:

```
map.entrySet().removeIf(e -> e.getValue().equals("mysql"));
```

Дополнительные сведения об удалении из коллекции `Map` можно найти в разд. 118 *"Удаление всех элементов коллекции, которые совпадают с предикатом"*.

113. Замена элементов в отображении *Map*

Задача замены элементов в отображении `Map` встречается часто, и с ней можно столкнуться в самых разных ситуациях. Удобное ее решение, избегающее спагетти-кода (т. е. запутанного кода), располагающегося во вспомогательном методе, опирается на JDK 8 и его метод `replace()`.

Допустим, что у нас есть класс `Melon` и отображение `Map`, состоящее из объектов класса `Melon`:

```
public class Melon {
    private final String type;
    private final int weight;

    // конструктор, геттеры, equals(), hashCode(),
    // toString() опущены для краткости
}
```

```
Map<Integer, Melon> mapOfMelon = new HashMap<>();
mapOfMelon.put(1, new Melon("Apollo", 3000));
mapOfMelon.put(2, new Melon("Jade Dew", 3500));
mapOfMelon.put(3, new Melon("Cantaloupe", 1500));
```

Заменить объект Melon, соответствующий ключу 2, можно посредством метода `replace(K key, V value)`. Если замена является успешной, то этот метод вернет исходный объект Melon:

```
// дыня Jade Dew(3500g) была заменена
Melon melon = mapOfMelon.replace(2, new Melon("Gac", 1000));
```

Теперь отображение Map содержит следующие элементы:

```
1=Apollo(3000g), 2=Gac(1000g), 3=Cantaloupe(1500g)
```

Далее предположим, что мы хотим заменить элемент с ключом 1 и дыню Apollo (3000 г). Таким образом, для того чтобы получить успешную замену, дыня в обоих случаях должна быть одинаковой. Это можно выполнить посредством булева метода `replace(K key, V oldValue, V newValue)`. Этот метод основан на контракте метода `equals()` при сравнении заданных значений, поэтому класс Melon должен имплементировать метод `equals()`, иначе результат будет непредсказуемым:

```
// true
boolean melon = mapOfMelon.replace(
    1, new Melon("Apollo", 3000), new Melon("Bitter", 4300));
```

Теперь отображение Map содержит следующие элементы:

```
1=Bitter(4300g), 2=Gac(1000g), 3=Cantaloupe(1500g)
```

Наконец, допустим, что мы хотим заменить все элементы в отображении Map, основываясь на заданной функции. Это можно сделать посредством метода `void replaceAll(BiFunction<? super K,? super V,? extends V> function)`.

Например, заменим все дыни, которые весят более 1000 г, дынями весом, равным 1000 г. Следующая ниже бифункция BiFunction формирует эту функцию (k — это ключ, v — значение каждого элемента отображения Map):

```
BiFunction<Integer, Melon, Melon> function = (k, v)
    -> v.getWeight() > 1000 ? new Melon(v.getType(), 1000) : v;
```

Затем на сцене появляется метод `replaceAll()`:

```
mapOfMelon.replaceAll(function);
```

Теперь отображение Map содержит следующие элементы:

```
1=Bitter(1000g), 2=Gac(1000g), 3=Cantaloupe(1000g)
```

114. Сравнение двух отображений Map

Сравнить два отображения Map между собой очень просто, если опираться на метод `Map.equals()`. Во время сравнения двух отображений Map этот метод сопоставляет их ключи и значения, используя метод `Object.equals()`.

Например, рассмотрим два отображения Map с объектами Melon, имеющими одинаковые элементы (наличие equals() и hashCode() является обязательным в классе Melon):

```
public class Melon {
    private final String type;
    private final int weight;

    // конструктор, геттеры, equals(), hashCode(),
    // toString() опущены для краткости
}
```

```
Map<Integer, Melon> melons1Map = new HashMap<>();
Map<Integer, Melon> melons2Map = new HashMap<>();
melons1Map.put(1, new Melon("Apollo", 3000));
melons1Map.put(2, new Melon("Jade Dew", 3500));
melons1Map.put(3, new Melon("Cantaloupe", 1500));
melons2Map.put(1, new Melon("Apollo", 3000));
melons2Map.put(2, new Melon("Jade Dew", 3500));
melons2Map.put(3, new Melon("Cantaloupe", 1500));
```

Теперь, если мы проверим melons1Map и melons2Map на эквивалентность, то получим true:

```
boolean equals12Map = melons1Map.equals(melons2Map); // true
```

Но это не сработает, если мы будем использовать массивы. Например, рассмотрим следующие два отображения Map:

```
Melon[] melons1Array = {
    new Melon("Apollo", 3000),
    new Melon("Jade Dew", 3500), new Melon("Cantaloupe", 1500)
};
```

```
Melon[] melons2Array = {
    new Melon("Apollo", 3000),
    new Melon("Jade Dew", 3500), new Melon("Cantaloupe", 1500)
};
```

```
Map<Integer, Melon[]> melons1ArrayMap = new HashMap<>();
melons1ArrayMap.put(1, melons1Array);
Map<Integer, Melon[]> melons2ArrayMap = new HashMap<>();
melons2ArrayMap.put(1, melons2Array);
```

Даже если melons1ArrayMap и melons2ArrayMap являются эквивалентными, метод Map.equals() вернет false:

```
boolean equals12ArrayMap = melons1ArrayMap.equals(melons2ArrayMap);
```

Проблема заключается в том, что метод equals() массива сравнивает не содержимое массива, а идентичность. Для того чтобы решить эту проблему, мы можем написать

вспомогательный метод следующим образом (на этот раз опираясь на метод `Arrays.equals()`, который сравнивает содержимое массивов):

```
public static <A, B> boolean equalsWithArrays(
    Map<A, B[]> first, Map<A, B[]> second) {

    if (first.size() != second.size()) {
        return false;
    }

    return first.entrySet().stream()
        .allMatch(e -> Arrays.equals(e.getValue(),
            second.get(e.getKey())));
}
```

115. Сортировка отображения *Map*

Существует несколько способов сортировки отображения `Map`. Для начала допустим, что у нас есть отображение `Map`, содержащее объекты `Melon`:

```
public class Melon implements Comparable {
    private final String type;
    private final int weight;

    @Override
    public int compareTo(Object o) {
        return Integer.compare(this.getWeight(), ((Melon) o).getWeight());
    }

    // конструктор, геттеры, equals(), hashCode(),
    // toString() для краткости опущены
}
```

```
Map<String, Melon> melons = new HashMap<>();
melons.put("delicious", new Melon("Apollo", 3000));
melons.put("refreshing", new Melon("Jade Dew", 3500));
melons.put("famous", new Melon("Cantaloupe", 1500));
```

Теперь рассмотрим несколько вариантов решения задачи сортировки этого отображения `Map`. Цель состоит в том, чтобы выставить наружу методы, перечисленные на рис. 5.7, посредством служебного класса с именем `Maps`.

● <code>sortByKeyList</code> (<code>Map<K, V> map</code>)	<code>List<K></code>
● <code>sortByKeyStream</code> (<code>Map<K, V> map</code> , <code>Comparator<? super K> c</code>)	<code>Map<K, V></code>
● <code>sortByKeyTreeMap</code> (<code>Map<K, V> map</code>)	<code>TreeMap<K, V></code>
● <code>sortByValueList</code> (<code>Map<K, V> map</code>)	<code>List<V></code>
● <code>sortByValueStream</code> (<code>Map<K, V> map</code> , <code>Comparator<? super V> c</code>)	<code>Map<K, V></code>

Рис. 5.7

Давайте рассмотрим разные варианты решения в следующих далее разделах.

Сортировка по ключу посредством *TreeMap* и в естественном порядке

Быстрое решение для сортировки отображения Map опирается на класс *TreeMap*. По определению ключи в классе *TreeMap* (древовидное отображение) сортируются по их естественному порядку. Кроме того, класс *TreeMap* имеет конструктор типа *TreeMap*(Map<? extends K, ? extends V> m):

```
public static <K, V> TreeMap<K, V> sortByKeyTreeMap(Map<K, V> map) {
    return new TreeMap<>(map);
}
```

И вызов его будет сортировать отображение по ключу:

```
// {delicious=Apollo(3000g),
// famous=Cantaloupe(1500g), refreshing=Jade Dew(3500g)}
TreeMap<String, Melon> sortedMap = Maps.sortByKeyTreeMap(melons);
```

Сортировка по ключу и значению посредством *Stream* и *Comparator*

После того как мы создадим *Stream* для отображения Map, мы можем легко отсортировать его посредством метода *Stream.sorted()* с компаратором или без него. На этот раз будем использовать *Comparator*:

```
public static <K, V> Map<K, V> sortByKeyStream(
    Map<K, V> map, Comparator<? super K> c) {

    return map.entrySet()
        .stream()
        .sorted(Map.Entry.comparingByKey(c))
        .collect(toMap(Map.Entry::getKey, Map.Entry::getValue,
            (v1, v2) -> v1, LinkedHashMap::new));
}
```

```
public static <K, V> Map<K, V> sortByValueStream(
    Map<K, V> map, Comparator<? super V> c) {

    return map.entrySet()
        .stream()
        .sorted(Map.Entry.comparingByValue(c))
        .collect(toMap(Map.Entry::getKey, Map.Entry::getValue,
            (v1, v2) -> v1, LinkedHashMap::new));
}
```

Нам нужно опираться на связное хеш-отображение *LinkedHashMap* вместо просто хеш-отображения *HashMap*. В противном случае мы не сможем сохранить порядок итераций.

Давайте отсортируем наше отображение следующим образом:

```
// {delicious=Apollo(3000g),  
// famous=Cantaloupe(1500g),  
// refreshing=Jade Dew(3500g)}  
Comparator<String> byInt = Comparator.naturalOrder();  
Map<String, Melon> sortedMap = Maps.sortByKeyStream(melons, byInt);  
  
// {famous=Cantaloupe(1500g),  
// delicious=Apollo(3000g),  
// refreshing=Jade Dew(3500g)}  
Comparator<Melon> byWeight = Comparator.comparing(Melon::getWeight);  
Map<String, Melon> sortedMap  
    = Maps.sortByValueStream(melons, byWeight);
```

Сортировка по ключу и значению посредством *List*

Предыдущие примеры сортируют заданное отображение `Map`, и результат также является отображением. Если нам нужны лишь отсортированные ключи (и нас не интересуют значения) или наоборот, то мы можем опереться на список, создаваемый посредством метода `Map.keySet()` для ключей и с помощью метода `Map.values()` для значений:

```
public static <K extends Comparable, V> List<K>  
    sortByKeyList(Map<K, V> map) {  
  
    List<K> list = new ArrayList<>(map.keySet());  
    Collections.sort(list);  
  
    return list;  
}
```

```
public static <K, V extends Comparable> List<V>  
    sortByValueList(Map<K, V> map) {  
  
    List<V> list = new ArrayList<>(map.values());  
    Collections.sort(list);  
  
    return list;  
}
```

А теперь давайте отсортируем наше отображение:

```
// [delicious, famous, refreshing]  
List<String> sortedKeys = Maps.sortByKeyList(melons);  
  
// [Cantaloupe(1500g), Apollo(3000g), Jade Dew(3500g)]  
List<Melon> sortedValues = Maps.sortByValueList(melons);
```

Если повторяющиеся значения не разрешены, то вы должны опереться на имплементацию с использованием отсортированного множества `SortedSet`:

```
SortedSet<String> sortedKeys = new TreeSet<>(melons.keySet());
SortedSet<Melon> sortedValues = new TreeSet<>(melons.values());
```

116. Копирование отображения *HashMap*

Удобный вариант решения для выполнения мелкой копии отображения `HashMap` опирается на конструктор `HashMap`: `HashMap(Map<? extends K,? extends V> m)`. Следующий фрагмент кода не требует пояснений:

```
Map<K, V> mapToCopy = new HashMap<>();
Map<K, V> shallowCopy = new HashMap<>(mapToCopy);
```

Еще один вариант решения может опираться на метод `putAll(Map<? extends K,? extends V> m)`. Этот метод копирует все попарные связки "ключ–значение" из указанного отображения в это отображение, как показано в следующем вспомогательном методе:

```
@SuppressWarnings("unchecked")
public static <K, V> HashMap<K, V> shallowCopy(Map<K, V> map) {
    HashMap<K, V> copy = new HashMap<>();
    copy.putAll(map);

    return copy;
}
```

Мы также можем написать этот вспомогательный метод в функциональном стиле Java 8 следующим образом:

```
@SuppressWarnings("unchecked")
public static <K, V> HashMap<K, V> shallowCopy(Map<K, V> map) {
    Set<Entry<K, V>> entries = map.entrySet();
    HashMap<K, V> copy = (HashMap<K, V>) entries.stream()
        .collect(Collectors.toMap(
            Map.Entry::getKey, Map.Entry::getValue));

    return copy;
}
```

Однако эти три варианта решения дают лишь мелкую копию отображения. Вариант решения для получения глубокой копии может опираться на библиотеку клонирования `Cloning` (<https://github.com/kostaskougios/cloning>), представленную в главе 2. Вспомогательный метод, который будет использовать клонирование, можно написать следующим образом:

```
@SuppressWarnings("unchecked")
public static <K, V> HashMap<K, V> deepCopy(Map<K, V> map) {
    Cloner cloner = new Cloner();
    HashMap<K, V> copy = (HashMap<K, V>) cloner.deepClone(map);

    return copy;
}
```


117. Слияние двух отображений *Map*

Слияние двух отображений *Map* представляет собой процесс соединения двух отображений в общее отображение, содержащее элементы обоих отображений. Кроме того, в случае коллизий ключей в итоговое отображение мы встраиваем значение, принадлежащее второму отображению. Но такое техническое решение является проектным.

Рассмотрим два следующих отображения *Map* (мы намеренно добавили коллизию для ключа 3):

```
public class Melon {
    private final String type;
    private final int weight;

    // конструктор, геттеры, equals(), hashCode(),
    // toString() опущены для краткости
}
```

```
Map<Integer, Melon> melons1 = new HashMap<>();
Map<Integer, Melon> melons2 = new HashMap<>();
melons1.put(1, new Melon("Apollo", 3000));
melons1.put(2, new Melon("Jade Dew", 3500));
melons1.put(3, new Melon("Cantaloupe", 1500));
melons2.put(3, new Melon("Apollo", 3000));
melons2.put(4, new Melon("Jade Dew", 3500));
melons2.put(5, new Melon("Cantaloupe", 1500));
```

Начиная с JDK 8, у нас есть метод в интерфейсе *Map*: *V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)*.

Если заданный ключ (*key*) не ассоциирован со значением либо ассоциирован с *null*, то новое значение будет равно *value*. Если заданный ключ (*key*) ассоциирован с непустым значением (не *null*), то новое значение вычисляется на основе заданной бифункции *BiFunction*. Если результат этой функции равен *null*, и ключ в отображении присутствует, то этот элемент будет из отображения удален.

Основываясь на этом определении, мы можем написать вспомогательный метод для слияния двух отображений следующим образом:

```
public static <K, V> Map<K, V> mergeMaps(
    Map<K, V> map1, Map<K, V> map2) {

    Map<K, V> map = new HashMap<>(map1);

    map2.forEach(
        (key, value) -> map.merge(key, value, (v1, v2) -> v2));

    return map;
}
```

Обратите внимание, что мы не модифицируем исходные отображения. Мы предпочитаем возвращать новое отображение, содержащее элементы первого отображения, объединенные с элементами второго отображения. В случае коллизии ключей мы заменяем существующее значение значением из второго отображения (v2).

Еще один вариант решения может быть написан на основе метода `Stream.concat()`. В сущности, этот метод конкатенирует два потока в один. Для того чтобы создать поток `Stream` из отображения `Map`, мы вызываем метод `Map.entrySet().stream()`. После слияния двух потоков, созданных на основе заданных отображений, мы просто собираем результат посредством коллектора `toMap()`:

```
public static <K, V> Map<K, V> mergeMaps(
    Map<K, V> map1, Map<K, V> map2) {

    Stream<Map.Entry<K, V>> combined
        = Stream.concat(map1.entrySet().stream(),
            map2.entrySet().stream());

    Map<K, V> map = combined.collect(
        Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue,
            (v1, v2) -> v2));

    return map;
}
```

В качестве бонуса множество `Set` (например, множество целых чисел) может быть отсортировано следующим образом:

```
List<Integer> sortedList = someSetOfIntegers.stream()
    .sorted().collect(Collectors.toList());
```

Для объектов следует опираться на метод `sorted(Comparator<? super T>)`.

118. Удаление всех элементов коллекции, которые совпадают с предикатом

В нашей коллекции будет находиться несколько объектов класса `Melon`:

```
public class Melon {
    private final String type;
    private final int weight;

    // конструктор, геттеры, equals(), hashCode(),
    // toString() для краткости опущены
}
```

Допустим, что у нас есть представленная ниже коллекция (список `ArrayList`), которая на протяжении всех наших примеров демонстрирует то, как можно удалять из нее элементы, совпадающие с заданным предикатом:

```
List<Melon> melons = new ArrayList<>();
melons.add(new Melon("Apollo", 3000));
```

```
melons.add(new Melon("Jade Dew", 3500));
melons.add(new Melon("Cantaloupe", 1500));
melons.add(new Melon("Gac", 1600));
melons.add(new Melon("Hami", 1400));
```

Давайте рассмотрим разные варианты решения, приведенные в следующих далее разделах.

Удаление посредством итератора

Удаление посредством `Iterator` является самым старым подходом из имеющихся в Java. Итератор позволяет нам обходить (или пересекать) коллекцию в цикле и удалять те или иные элементы. Этот самый старый подход также имеет некоторые недостатки. Прежде всего, в зависимости от типа коллекции удаление посредством итератора подвержено возникновению исключения `ConcurrentModificationException`, если несколько нитей исполнения модифицируют коллекцию. Кроме того, удаление не ведет себя одинаково для всех коллекций (например, удаление из связного списка `LinkedList` происходит быстрее, чем удаление из списка `ArrayList`, потому что первый просто перемещает указатель на следующий элемент, в то время как второй должен сдвинуть все элементы). Тем не менее этот вариант решения представлен в прилагаемом исходном коде.

Если вам требуется только размер итерируемого объекта `Iterable`, то рассмотрим один из следующих подходов:

```
// для любого итерируемого объекта (Iterable)
StreamSupport.stream(iterable.splititerator(), false).count();

// для коллекций
((Collection<?>) iterable).size()
```

Удаление посредством `Collection.removeIf()`

Начиная с JDK 8, мы можем сократить приведенный выше код до одной строки кода посредством метода `Collection.removeIf()`. Этот метод опирается на предикат, как показано в следующем ниже примере:

```
melons.removeIf(t -> t.getWeight() < 3000);
```

На этот раз объект `ArrayList` обходит список в цикле и помечает для удаления те элементы, которые удовлетворяют нашему предикату `Predicate`. Далее `ArrayList` снова выполняет обход, чтобы удалить помеченные и сдвинуть оставшиеся элементы.

Используя этот подход, списки `LinkedList` и `ArrayList` работают практически одинаково.

Удаление посредством `Stream`

Начиная с JDK 8, мы можем создать поток из коллекции (`Collection.stream()`) и отфильтровать его элементы посредством метода `filter(Predicate p)`. Фильтр оставит только те элементы, которые удовлетворяют заданному предикату.

Наконец, мы собираем эти элементы посредством соответствующего коллектора:

```
List<Melon> filteredMelons = melons.stream()
    .filter(t -> t.getWeight() >= 3000)
    .collect(Collectors.toList());
```



В отличие от двух других вариантов решения, этот вариант не мутирует исходную коллекцию, но может работать медленнее и потреблять больше памяти.

Разбиение элементов посредством коллектора *Collectors.partitioningBy()*

Иногда не нужно удалять элементы, которые не совпадают с нашим предикатом. Требуется только выделить элементы на основе нашего предиката. Так вот, это достижимо посредством коллектора `Collectors.partitioningBy(Predicate p)`.

Коллектор `Collectors.partitioningBy()` разделяет элементы на два списка, которые добавляются в отображение `Map` в качестве значений. Два ключа этого отображения будут равны `true` и `false`:

```
Map<Boolean, List<Melon>> separatedMelons = melons.stream()
    .collect(Collectors.partitioningBy(
        (Melon t) -> t.getWeight() >= 3000));
```

```
List<Melon> weightLessThan3000 = separatedMelons.get(false);
List<Melon> weightGreaterThan3000 = separatedMelons.get(true);
```

Таким образом, ключ `true` используется для получения списка элементов, совпадающих с предикатом, а ключ `false` — для получения списка элементов, не совпадающих с предикатом.

В качестве бонуса, если мы хотим проверить, что все элементы коллекции `List` являются одинаковыми, мы можем опереться на метод `Collections.frequency(Collection c, Object obj)`. Он возвращает число элементов в данной коллекции, эквивалентных указанному объекту:

```
boolean allTheSame = Collections.frequency(
    melons, melons.get(0)) == melons.size();
```

Если переменная `allTheSame` равна `true`, то все элементы являются эквивалентными. Обратите внимание, что методы `equals()` и `hashCode()` объекта из коллекции `List` должны быть имплементированы соответствующим образом.

119. Конвертирование коллекции в массив

В конвертировании коллекции в массив мы можем опереться на метод `Collection.toArray()`. Без аргументов он конвертирует заданную коллекцию в `Object[]`, как в следующем примере:

```
List<String> names = Arrays.asList("ana", "mario", "vio");
Object[] namesArrayAsObjects = names.toArray();
```

Очевидно, что это не совсем полезно, поскольку вместо `Object[]` мы ожидаем `String[]`. Это можно достичь посредством метода `Collection.toArray(T[] a)` следующим образом:

```
String[] namesArraysAsStrings = names.toArray(new
String[names.size()]);
String[] namesArraysAsStrings = names.toArray(new String[0]);
```

Из этих двух вариантов решения второе более предпочтительно, т. к. мы избегаем вычисления размера коллекции.

Но начиная с JDK 11, существует еще один метод, посвященный этой операции, — `Collection.toArray(IntFunction<T[]> generator)`. Он возвращает массив, содержащий все элементы в этой коллекции, используя генераторную функцию, предусмотренную для выделения пространства под возвращаемый массив:

```
String[] namesArraysAsStrings = names.toArray(String[]::new);
```

Помимо модифицируемого списка фиксированного размера, создаваемого с помощью метода `Arrays.asList()`, мы можем построить немодифицируемую коллекцию `List/Set` из массива посредством методов `of()`:

```
String[] namesArray = {"ana", "mario", "vio"};
```

```
List<String> namesArrayAsList = List.of(namesArray);
```

```
Set<String> namesArrayAsSet = Set.of(namesArray);
```

120. Фильтрация коллекции по списку

Часто встречающейся задачей, с которой мы сталкиваемся в приложениях, является фильтрация коллекции по списку. В основном мы начинаем с огромной коллекции `Collection` и хотим извлечь из нее элементы, которые совпадают с элементами списка.

В следующих ниже примерах давайте рассмотрим класс `Melon`:

```
public class Melon {
    private final String type;
    private final int weight;

    // конструктор, геттеры, equals(), hashCode(),
    // toString() опущены для краткости
}
```

Здесь мы имеем огромную коллекцию (в данном случае список `ArrayList`) объектов класса `Melon`:

```
List<Melon> melons = new ArrayList<>();
melons.add(new Melon("Apollo", 3000));
melons.add(new Melon("Jade Dew", 3500));
melons.add(new Melon("Cantaloupe", 1500));
melons.add(new Melon("Gac", 1600));
melons.add(new Melon("Hami", 1400));
...
```

И у нас также есть список, содержащий сорта дынь, которые мы хотим извлечь из приведенного выше списка `ArrayList`:

```
List<String> melonsByType
    = Arrays.asList("Apollo", "Gac", "Crenshaw", "Hami");
```

Один из вариантов решения этой задачи может предусматривать обход обеих коллекций в цикле и сравнение сортов дыни, но результирующий код будет довольно медленным. Еще один вариант решения этой задачи может привлекать метод `List.contains()` и лямбда-выражение:

```
List<Melon> results = melons.stream()
    .filter(t -> melonsByType.contains(t.getType()))
    .collect(Collectors.toList());
```

Этот код является компактным и быстрым. За кулисами, метод `List.contains()` опирается на следующую проверку:

```
// размер - это размер melonsByType
// o - текущий элемент, искомый в melons
// elementData - melonsByType
for (int i = 0; i < size; i++)
    if (o.equals(elementData[i])) {
        return i;
    }
}
```

Однако мы можем придать еще один импульс росту производительности с помощью варианта решения, который опирается на метод `HashSet.contains()` вместо метода `List.contains()`. В отличие от метода `List.contains()`, который использует приведенную выше инструкцию `for` для сопоставления элементов, метод `HashSet.contains()` вызывает метод `Map.containsKey()`. В общих чертах, множество `Set` имплементировано на основе отображения `Map`, и каждый добавленный элемент отображается как "ключ–значение" типа элемент-ПРИСУТСТВУЕТ. Таким образом, элемент является ключом в этом отображении, в то время как ПРИСУТСТВУЕТ — это просто фиктивное значение.

Когда мы вызываем метод `HashSet.contains(element)`, мы фактически вызываем `Map.containsKey(element)`. Этот метод сопоставляет заданный элемент с соответствующим ключом в отображении на основе его `hashCode()`, что намного быстрее, чем `equals()`.

После того как мы конвертируем исходный список `ArrayList` в хеш-множество `HashSet`, мы готовы пойти дальше:

```
Set<String> melonsSetByType = melonsByType.stream()
    .collect(Collectors.toSet());

List<Melon> results = melons.stream()
    .filter(t -> melonsSetByType.contains(t.getType()))
    .collect(Collectors.toList());
```

К тому же это решение работает быстрее, чем предыдущее. Оно должно выполняться за половину времени, требуемого предыдущим решением.

121. Замена элементов в списке

Еще одна часто встречающаяся задача, с которой мы сталкиваемся в приложениях, заключается в замене элементов списка, совпадающих с теми или иными условиями.

В следующем примере возьмем класс `Melon`:

```
public class Melon {
    private final String type;
    private final int weight;

    // конструктор, геттеры, equals(), hashCode(),
    // toString() опущены для краткости
}
```

И рассмотрим список объектов класса `Melon`:

```
List<Melon> melons = new ArrayList<>();
```

```
melons.add(new Melon("Apollo", 3000));
melons.add(new Melon("Jade Dew", 3500));
melons.add(new Melon("Cantaloupe", 1500));
melons.add(new Melon("Gac", 1600));
melons.add(new Melon("Hami", 1400));
```

Допустим, что мы хотим заменить все дыни весом менее 3000 г другими дынями того же сорта, которые весят 3000 г.

Решение этой задачи потребует обхода списка в цикле, а затем использования метода `List.set(int index, E element)` для замены дынь соответствующим образом.

Следующий ниже фрагмент кода является примером спагетти-кода:

```
for (int i = 0; i < melons.size(); i++) {
    if (melons.get(i).getWeight() < 3000) {
        melons.set(i, new Melon(melons.get(i).getType(), 3000));
    }
}
```

Еще одно решение опирается на функциональный стиль Java 8 или, точнее, на функциональный интерфейс `UnaryOperator`.

Основываясь на этом функциональном интерфейсе, мы можем написать вот такой оператор:

```
UnaryOperator<Melon> operator = t
    -> (t.getWeight() < 3000) ? new Melon(t.getType(), 3000) : t;
```

Теперь мы можем использовать метод `List.replaceAll(UnaryOperator<E> operator)` JDK 8 следующим образом:

```
melons.replaceAll(operator);
```

Оба подхода должны работать практически одинаково.

122. Нитебезопасные коллекции, стеки и очереди

Всякий раз, когда возникает вероятность обращения к коллекции/стеку/очереди из нескольких нитей исполнения, такие коллекция/стек/очередь также подвержены возникновению исключений, специфичных для конкурентного доступа (например, `java.util.ConcurrentModificationException`). Теперь давайте познакомимся с кратким обзором и введением во встроенные в Java конкурентные коллекции.

Конкурентные коллекции

К счастью, Java предоставляет нитебезопасные (конкурентные) альтернативы ненитебезопасным коллекциям (включая стеки и очереди), как показано ниже.

Нитебезопасные списки

Нитебезопасной версией класса `ArrayList` является класс `CopyOnWriteArrayList`. В табл. 5.1 перечислены встроенные в Java однопоточные и многопоточные списки.

Таблица 5.1

Однопоточный	Многопоточный
<code>ArrayList</code>	<code>CopyOnWriteArrayList</code> (частые чтения, редкие обновления)
<code>LinkedList</code>	<code>Vector</code>

Имплементация списка `CopyOnWriteArrayList` содержит элементы в массиве. Всякий раз, когда мы вызываем метод, который мутирует список (например, `add()`, `set()` и `remove()`), Java будет работать с копией этого массива.

Итератор над этой коллекцией будет оперировать немутуируемой копией коллекции. Таким образом, исходная коллекция может быть модифицирована без проблем. Потенциальные модификации исходной коллекции не видны в итераторе:

```
List<Integer> list = new CopyOnWriteArrayList<>();
```



Эту коллекцию следует использовать, когда чтение происходит часто, а изменения — редко.

Нитебезопасное множество

Нитебезопасной версией класса `Set` является класс `CopyOnWriteArraySet`. В табл. 5.2 перечислены встроенные в Java однопоточные и многопоточные множества.

Таблица 5.2

Однопоточное	Многопоточное
• <code>HashSet</code>	• <code>ConcurrentSkipListSet</code> (сортированное множество)

Таблица 5.2 (окончание)

Однонитевое	Многонитевое
• TreeSet (сортированное множество) • LinkedHashSet (поддержка порядка вставок) • BitSet • EnumSet	• CopyOnWriteArraySet (частые чтения, редкие обновления)

Set — это такое множество, в котором для всех своих операций используется внутренний список CopyOnWriteArrayList. Создать такое множество Set можно следующим образом:

```
Set<Integer> set = new CopyOnWriteArraySet<>();
```



Эту коллекцию следует использовать, когда чтения происходят часто, а изменения — редко.

Нитебезопасной версией класса NavigableSet является класс ConcurrentSkipListSet (конкурентная имплементация коллекции SortedSet, в которой наиболее базовые операции имеют сложность $O(\log n)$).

Нитебезопасное отображение Map

Нитебезопасной версией отображения Map является отображение ConcurrentHashMap. В табл. 5.3 перечислены встроенные в Java однонитевые и многонитевые отображения.

Таблица 5.3

Однонитевое	Многонитевое
• HashMap • TreeMap (сортированные ключи) • LinkedHashMap (поддержка порядка вставок) • IdentityHashMap (ключи сравниваются посредством ==) • WeakHashMap • EnumMap	• ConcurrentHashMap • ConcurrentSkipListMap (сортированное отображение) • Hashtable

Отображение ConcurrentHashMap позволяет выполнять операции извлечения (например, get()) без блокирования. Это означает, что операции извлечения могут накладываться с операциями обновления (включая put() и remove()). Создать отображение ConcurrentHashMap можно следующим образом:

```
ConcurrentMap<Integer, Integer> map = new ConcurrentHashMap<>();
```



Всякий раз, когда требуются нитебезопасность и высокая производительность, вы можете опереться на нитебезопасную версию отображения Map, т. е. на отображение `ConcurrentHashMap`. Следует избегать коллекций `Hashtable` и `Collections.synchronizedMap()`, т. к. у них плохая производительность.

Для отображения `ConcurrentMap`, поддерживающего `NavigableMap`, операции опираются на отображение `ConcurrentSkipListMap`:

```
ConcurrentNavigableMap<Integer, Integer> map
    = new ConcurrentSkipListMap<>();
```

Нитебезопасная очередь, поддерживаемая массивом

Java предоставляет нитебезопасную очередь (первым вошел, первым вышел — *first in, first out, FIFO*), поддерживаемую массивом посредством класса `ArrayBlockingQueue`. В табл. 5.4 перечислены встроенные в Java однопоточные и многопоточные очереди, поддерживаемые массивом.

Таблица 5.4

Однопоточный	Многопоточный
• <code>ArrayDeque</code> • <code>PriorityQueue</code> (сортированные извлечения)	• <code>ArrayBlockingQueue</code> (ограниченная) • <code>ConcurrentLinkedQueue</code> (неограниченная) • <code>ConcurrentLinkedDeque</code> (неограниченная) • <code>LinkedBlockingQueue</code> (опционально ограниченная) • <code>LinkedBlockingDeque</code> (опционально ограниченная) • <code>LinkedTransferQueue</code> • <code>PriorityBlockingQueue</code> • <code>SynchronousQueue</code> • <code>DelayQueue</code> • <code>Stack</code>

Емкость очереди `ArrayBlockingQueue` не может быть изменена после создания. Попытки поместить элемент в полную очередь приведут к блокированию операции; попытки взять элемент из пустой очереди также будут блокироваться. Очередь `ArrayBlockingQueue` можно легко создать следующим образом:

```
BlockingQueue<Integer> queue = new
ArrayBlockingQueue<>(QUEUE_MAX_SIZE);
```



Java также идет в комплекте с двумя нитебезопасными, опционально ограниченными блокирующими очередями, основанными на связанных узлах посредством `LinkedBlockingQueue` и `LinkedBlockingDeque` (двухсторонняя очередь (*deque*) — это линейная коллекция, которая поддерживает вставку и удаление элементов с обоих концов).

Нитебезопасная очередь на основе связанных узлов

Java предусматривает неограниченную нитебезопасную очередь/двухстороннюю очередь, поддерживаемую связными узлами посредством классов `ConcurrentLinkedDeque`/`ConcurrentLinkedQueue`. Вот пример объявления двухсторонней очереди `ConcurrentLinkedDeque`:

```
Deque<Integer> queue = new ConcurrentLinkedDeque<>();
```

Нитебезопасные очереди с приоритетом

Java предусматривает неограниченную нитебезопасную блокирующую очередь с приоритетом, основанную на куче с приоритетами посредством класса `PriorityBlockingQueue`. Двухстороннюю очередь `PriorityBlockingQueue` можно легко создать следующим образом:

```
BlockingQueue<Integer> queue = new PriorityBlockingQueue<>();
```



Ее не-нитебезопасная версия называется `PriorityQueue`.

Нитебезопасная очередь задержки

Java предусматривает нитебезопасную неограниченную блокирующую очередь, в которой элемент может быть принят только тогда, когда его задержка истекла, посредством класса `DelayQueue`. Очередь `DelayQueue` так же легко создать, как показано ниже:

```
BlockingQueue<TrainDelay> queue = new DelayQueue<>();
```

Нитебезопасная трансферная очередь

Java предусматривает нитебезопасную неограниченную трансферную очередь, основанную на связанных узлах, посредством класса `LinkedTransferQueue`.

Это такая очередь с дисциплиной доступа "первым вошел, первым вышел" (first in first out, FIFO), в которой головой очереди является элемент, находившийся в очереди дольше всего для некоего производителя. Хвостом очереди является элемент, который находился в очереди самое короткое время для некоего производителя.

Один из способов создания очереди выглядит следующим образом:

```
TransferQueue<String> queue = new LinkedTransferQueue<>();
```

Нитебезопасные синхронные очереди

Java предусматривает блокирующую очередь, в которой каждая операция вставки должна ожидать соответствующей операции удаления другой нитью исполнения, и наоборот, посредством класса `SynchronousQueue`:

```
BlockingQueue<String> queue = new SynchronousQueue<>();
```

Нитебезопасный стек

Нитебезопасными имплементациями стека являются классы `Stack` и `ConcurrentLinkedDeque`.

Класс `Stack` представляет собой стек объектов с дисциплиной доступа "последним вошел, первым вышел" (last in first out, LIFO). Он расширяет класс `Vector` несколькими операциями, которые позволяют обрабатывать вектор как стек. Каждый метод стека синхронизирован. Создать стек просто:

```
Stack<Integer> stack = new Stack<>();
```

Имплементация двухсторонней очереди `ConcurrentLinkedDeque` может использоваться в качестве стека (LIFO) посредством ее методов `push()` и `pop()`:

```
Deque<Integer> stack = new ConcurrentLinkedDeque<>();
```



В целях более высокой производительности следует отдавать предпочтение двухсторонней очереди `ConcurrentLinkedDeque` над `Stack`.

Исходный код, прилагаемый к этой книге, идет в комплекте с приложением для каждой приведенной выше коллекции с охватом нескольких нитей исполнения с целью проявления их нитебезопасного характера.

Синхронизированные коллекции

Помимо конкурентных коллекций, у нас также есть синхронизированные коллекции. Java предоставляет набор оберток, которые выставляют коллекцию наружу как нитебезопасную коллекцию. Эти обертки доступны в `Collections`. Наиболее распространенными из них являются следующие:

◆ `synchronizedCollection(Collection<T> c)` — возвращает синхронизированную (нитебезопасную) коллекцию, поддерживаемую заданной коллекцией;

◆ `synchronizedList(List<T> list)` — возвращает синхронизированный (нитебезопасный) список, поддерживаемый заданным списком:

```
List<Integer> syncList
    = Collections.synchronizedList(new ArrayList<>());
```

◆ `synchronizedMap(Map<K, V> m)` — возвращает синхронизированное (нитебезопасное) отображение, поддерживаемое заданным отображением:

```
Map<Integer, Integer> syncMap
    = Collections.synchronizedMap(new HashMap<>());
```

◆ `synchronizedSet(Set<T> s)` — возвращает синхронизированное (нитебезопасное) множество, поддерживаемое заданным множеством:

```
Set<Integer> syncSet
    = Collections.synchronizedSet(new HashSet<>());
```

Конкурентные коллекции против синхронизированных

Возникает очевидный вопрос: *в чем разница между конкурентной и синхронизированной коллекциями?* Так вот, главное отличие состоит в том, каким образом они достигают нитебезопасности. Конкурентные коллекции обеспечивают нитебезопасность путем разделения данных на сегменты. С одной стороны, нити исполнения могут обращаться к этим сегментам конкурентно и приобретать замки только на тех сегментах, которые ими используются. С другой стороны, синхронизированная коллекция блокирует всю коллекцию целиком посредством *внутреннего замкового механизма* (нить исполнения, которая вызывает синхронизированный метод, автоматически приобретает внутренний замок для объекта этого метода и освобождает его, когда метод возвращается).

Обход синхронизированной коллекции в цикле требует ручной синхронизации следующим образом:

```
List syncList = Collections.synchronizedList(new ArrayList());
...
synchronized(syncList) {
    Iterator i = syncList.iterator();
    while (i.hasNext()) {
        do_something_with i.next();
    }
}
```



Поскольку конкурентные коллекции позволяют нитям исполнения осуществлять конкурентный доступ, они гораздо эффективнее, чем синхронизированная коллекция.

123. Поиск сперва в ширину

Поиск сперва в ширину (breadth-first search, BFS) — это классический алгоритм обхода (посещения) всех узлов графа или дерева.

Самый простой способ понять этот алгоритм — воспользоваться псевдокодом и примером. Псевдокод алгоритма поиска сперва в ширину выглядит следующим образом:

1. Создать очередь Q .
2. Пометить элемент v как посещенный и поставить v в Q .
3. До тех пор, пока Q не является пустой.
4. Удалить из Q голову h .
5. Пометить и поставить в очередь всех (непосещенных) соседей головы h .

Допустим, что у нас есть граф на **шаге 0** (рис. 5.8).

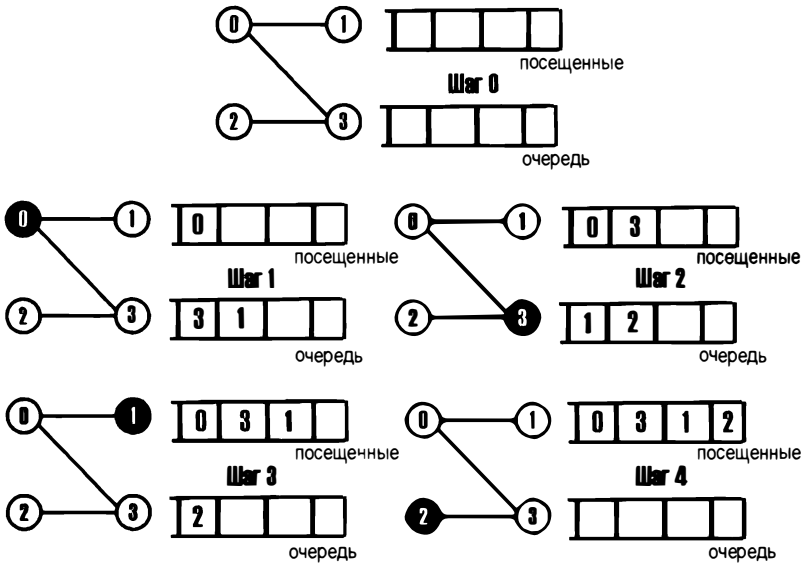


Рис. 5.8

На первом шаге (**шаг 1**) мы посещаем вершину 0. Мы помещаем ее в список посещенных вершин и все ее соседние вершины в очередь `queue` (3, 1). Далее на **шаге 2** мы посещаем элемент в начале очереди со значением 3. Вершина 3 имеет непосещенную смежную вершину в 2, поэтому мы добавляем ее в конец очереди. Далее, на **шаге 3**, мы посещаем элемент в начале очереди со значением 1. Эта вершина имеет одну смежную вершину (0), но она была посещена. Наконец, мы посещаем вершину 2, последнюю в очереди. Эта вершина имеет одну смежную вершину (3), которая уже была посещена.

В исходном коде алгоритм поиска сперва в ширину можно имплементировать следующим образом:

```
public class Graph {
    private final int v;
    private final LinkedList<Integer>[] adjacents;

    public Graph(int v) {

        this.v = v;
        adjacents = new LinkedList[v];

        for (int i = 0; i < v; ++i) {
            adjacents[i] = new LinkedList();
        }
    }

    public void addEdge(int v, int e) {
        adjacents[v].add(e);
    }
}
```

```
public void BFS(int start) {
    boolean visited[] = new boolean[v];
    LinkedList<Integer> queue = new LinkedList<>();
    visited[start] = true;

    queue.add(start);

    while (!queue.isEmpty()) {
        start = queue.poll();
        System.out.print(start + " ");

        Iterator<Integer> i = adjacents[start].listIterator();
        while (i.hasNext()) {
            int n = i.next();
            if (!visited[n]) {
                visited[n] = true;
                queue.add(n);
            }
        }
    }
}
```

И если мы введем следующий ниже граф (из приведенной выше схемы), то у нас будет следующее:

```
Graph graph = new Graph(4);
graph.addEdge(0, 3);
graph.addEdge(0, 1);
graph.addEdge(1, 0);
graph.addEdge(2, 3);
graph.addEdge(3, 0);
graph.addEdge(3, 2);
graph.addEdge(3, 3);
```

На выходе будут получены следующие данные: 0 3 1 2.

124. Префиксное дерево

Префиксное дерево (также именуемое цифровым деревом, или trie) — это упорядоченная древесная структура, обычно используемая для хранения строк. Его название происходит от того, что Trie является структурой данных, специально предназначенной для извлечения (reTrieval) данных. Его производительность лучше, чем у двоичного дерева.

За исключением корня префиксного дерева, каждый его узел содержит один символ (например, для слова heu будет три узла).

В общих чертах, каждый узел префиксного дерева содержит следующее:

- ◆ значение (символ или цифру);
- ◆ указатели на дочерние узлы;
- ◆ флаг, который равен `true`, если текущий узел завершает слово;
- ◆ один-единственный корень, используемый для ветвления узлов.

Схема на рис. 5.9 представляет последовательность шагов построения префиксного дерева, содержащего слова `cat`, `caret` и `bye`.

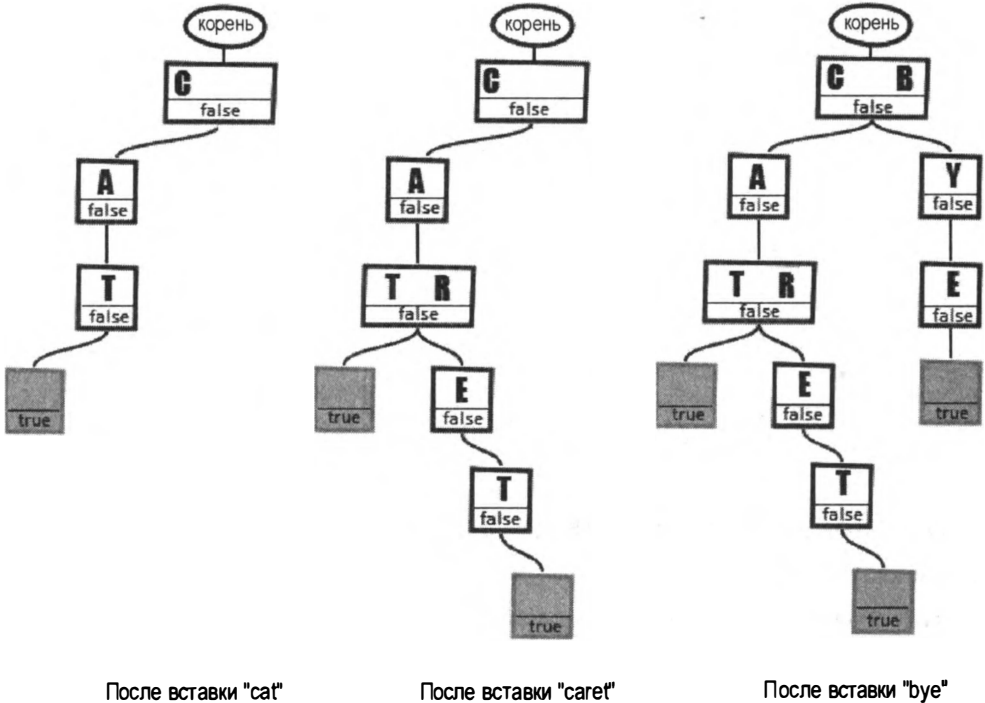


Рис. 5.9

В исходном коде узел префиксного дерева может быть сформирован так:

```
public class Node {
    private final Map<Character, Node> children = new HashMap<>();
    private boolean word;

    Map<Character, Node> getChildren() {
        return children;
    }

    public boolean isWord() {
        return word;
    }
}
```



```
public void setWord(boolean word) {
    this.word = word;
}
}
```

Основываясь на этом классе, мы можем определить базовую структуру префиксного дерева следующим образом:

```
class Trie {
    private final Node root;

    public Trie() {
        root = new Node();
    }

    public void insert(String word) {
        ...
    }

    public boolean contains(String word) {
        ...
    }

    public boolean delete(String word) {
        ...
    }
}
```

Вставка слова в префиксное дерево

Теперь давайте сосредоточимся на алгоритме вставки слов в префиксное дерево.

1. Рассматривать текущий узел как корень.
2. Перебрать символы заданного слова, начиная с первого символа.
3. Если текущий узел (`Map<Character, Node>`) отображает значение (`Node`) для текущего символа, то просто перейти к этому узлу. В противном случае создать новый узел, установить его символ, равным текущему символу, и перейти к этому узлу.
4. Повторять с шага 2 (перейти к следующему символу) до конца слова.
5. Пометить текущий узел как узел, который завершает слово.

В терминах исходного кода мы имеем следующее:

```
public void insert(String word) {
    Node node = root;

    for (int i = 0; i < word.length(); i++) {
        char ch = word.charAt(i);
        Function function = k -> new Node();
```

```

    node = node.getChildren().computeIfAbsent(ch, function);
}

node.setWord(true);
}

```



Сложность вставки равна $O(n)$, где n — это размер слова.

Отыскание слова в префиксном дереве

Теперь давайте отыщем слово в префиксном дереве.

1. Рассматривать текущий узел как корень.
2. Перебрать символы заданного слова (начать с первого символа).
3. По каждому символу проверить его присутствие в префиксном дереве (в `Map<Character, Node>`).
4. Если символ отсутствует, то вернуть `false`.
5. Повторять с шага 2 до конца слова.
6. В конце слова вернуть `true`, если это было слово, или `false`, если это был просто префикс.

В терминах исходного кода мы имеем следующее:

```

public boolean contains(String word) {
    Node node = root;

    for (int i = 0; i < word.length(); i++) {
        char ch = word.charAt(i);
        node = node.getChildren().get(ch);
        if (node == null) {
            return false;
        }
    }

    return node.isWord();
}

```



Сложность отыскания равна $O(n)$, где n — это размер слова.

Удаление слова из префиксного дерева

Наконец, давайте попробуем удалить слово из префиксного дерева.

1. Проверить, является ли заданное слово частью префиксного дерева.
2. Если оно является частью префиксного дерева, то просто его удалить.

Удаление происходит снизу вверх с помощью рекурсии и выполнения вот этих правил:

- ◆ если заданного слова в префиксном дереве нет, то ничего не происходит (вернуть false);
- ◆ если заданное слово является уникальным (не является частью другого слова), то удалить все соответствующие узлы (вернуть true);
- ◆ если заданное слово является префиксом другого длинного слова в префиксном дереве, то установить флаг листового узла равным false (вернуть false);
- ◆ если заданное слово имеет хотя бы еще одно слово в качестве префикса, то удалить соответствующие узлы с конца заданного слова до первого листового узла самого длинного префиксного слова (вернуть false).

В терминах исходного кода мы имеем следующее:

```
public boolean delete(String word) {
    return delete(root, word, 0);
}

private boolean delete(Node node, String word, int position) {
    if (word.length() == position) {
        if (!node.isWord()) {
            return false;
        }

        node.setWord(false);

        return node.getChildren().isEmpty();
    }

    char ch = word.charAt(position);
    Node children = node.getChildren().get(ch);

    if (children == null) {
        return false;
    }

    boolean deleteChildren = delete(children, word, position + 1);

    if (deleteChildren && !children.isWord()) {
        node.getChildren().remove(ch);

        return node.getChildren().isEmpty();
    }

    return false;
}
```



Сложность удаления равна $O(n)$, где n — это размер слова.

Теперь мы можем построить префиксное дерево следующим образом:

```
Trie trie = new Trie();
trie.insert/contains/delete(...);
```

125. Кортеж

В сущности, кортеж — это структура данных, состоящая из более чем одной части. Обычно кортеж состоит из двух или трех частей. В типичной ситуации, когда требуется более трех частей, выделенный класс является оптимальным вариантом выбора.

Кортежи являются немутуируемыми и используются всякий раз, когда нам нужно вернуть из метода несколько результатов. Например, допустим, что у нас есть метод, который возвращает минимум и максимум массива. Обычно метод не может возвращать оба сразу, и использование кортежа — удобное решение.

К сожалению, Java не обеспечивает встроенную поддержку кортежей. Тем не менее Java идет в комплекте со статическим интерфейсом `Map.Entry<K,V>`, который используется для представления элемента отображения. Кроме того, начиная с JDK 9, интерфейс `Map` был дополнен методом `entry(K k, V v)`, который возвращает немодифицируемый экземпляр `Map.Entry<K, V>`, содержащий заданные ключ и значение.

Для кортежа из двух частей мы можем написать наш метод следующим образом:

```
public static <T> Map.Entry<T, T> array(
    T[] arr, Comparator<? super T> c) {

    T min = arr[0];
    T max = arr[0];

    for (T elem: arr) {
        if (c.compare(min, elem) > 0) {
            min = elem;
        } else if (c.compare(max, elem) < 0) {
            max = elem;
        }
    }

    return entry(min, max);
}
```

Если этот метод располагается в классе с именем `Bounds`, мы можем вызвать его следующим образом:

```
public class Melon {
    private final String type;
    private final int weight;
```

```
// конструктор, геттеры, equals(), hashCode(),
// toString() опущены для краткости
}

Melon[] melons = {
    new Melon("Crenshaw", 2000), new Melon("Gac", 1200),
    new Melon("Bitter", 2200), new Melon("Hami", 800)
};

Comparator<Melon> byWeight = Comparator.comparing(Melon::getWeight);
Map.Entry<Melon, Melon> minmax = Bounds.array(melons, byWeight);

System.out.println("Min: " + minmax1.getKey()); // Hami (800g)
System.out.println("Max: " + minmax1.getValue()); // Bitter (2200g)
```

Но мы также можем написать имплементацию. Кортеж с двумя частями принято называть *парой*; поэтому интуитивная имплементация может быть следующей:

```
public final class Pair<L, R> {
    final L left;
    final R right;

    public Pair(L left, R right) {
        this.left = left;
        this.right = right;
    }

    static <L, R> Pair<L, R> of (L left, R right) {
        return new Pair<>(left, right);
    }

    // equals() и hashCode() опущены для краткости
}
```

Теперь мы можем переписать наш метод, который вычисляет минимум и максимум следующим образом:

```
public static <T> Pair<T, T> array(T[] arr, Comparator<? super T> c) {
    ...
    return Pair.of(min, max);
}
```

126. Структура данных Union Find

Алгоритм объединения и поиска (Union Find) оперирует структурой данных, состоящей из *дизъюнктивных множеств*.

Структура данных дизъюнктивных множеств определяет множества элементов, разделенных на некие непересекающиеся подмножества, которые не накладываются

ся друг на друга. Графически мы можем представить дизъюнктивное множество тремя подмножествами, как показано на рис. 5.10.

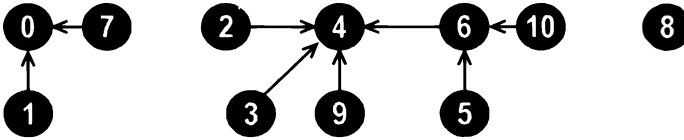


Рис. 5.10

В исходном коде дизъюнктивное множество представляется следующим образом:

- ◆ n — это общее число элементов (например, на приведенной выше схеме n равно 11);
- ◆ $rank$ — это массив, инициализированный нулями, который полезен для определения способа объединения двух подмножеств с многочисленными элементами (подмножества с более низким рангом $rank$ становятся детьми подмножеств с более высоким рангом $rank$);
- ◆ $parent$ — это массив, который позволяет нам строить структуру данных Union Find на основе массива (изначально $parent[0] = 0$; $parent[1] = 1$; ... $parent[10] = 10$);

```

public DisjointSet(int n) {
    this.n = n;
    rank = new int[n];
    parent = new int[n];

    initializeDisjointSet();
}
    
```

Алгоритмы поиска-объединения должны уметь:

- ◆ объединять два подмножества в одно подмножество;
- ◆ для заданного элемента возвращать его подмножества (это полезно для поиска элементов, которые находятся в том же подмножестве).

Для того чтобы сохранить в памяти дизъюнктивную структуру данных, мы можем представить ее в виде массива. Первоначально, в каждом индексе массива, мы храним вот этот индекс: $x[i] = i$. Каждый индекс может быть сопоставлен с частью значимой для нас информации, но это не обязательно. Например, такой массив может быть сформирован, как показано на рис. 5.11 (изначально у нас есть 11 подмножеств, и каждый элемент является родителем самого себя).

Если мы используем числа, то мы можем представить его так, как показано на рис. 5.12.

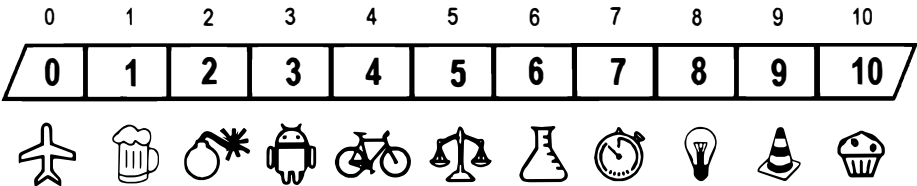


Рис. 5.11

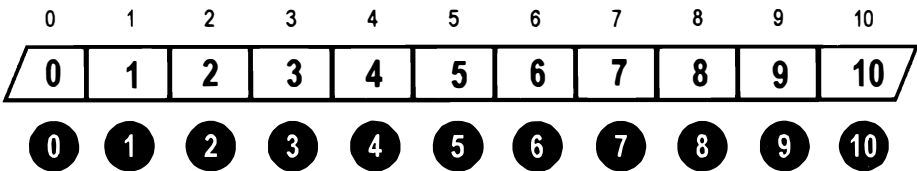


Рис. 5.12

В терминах исходного кода мы имеем следующее:

```
private void initializeDisjointSet() {  
    for (int i = 0; i < n; i++) {  
        parent[i] = i;  
    }  
}
```

Далее мы должны определить наши подмножества посредством операции *объединения*. Мы можем определить подмножества с помощью последовательности пар (*родитель, ребенок*). Для примера определим три пары — `union(0, 1);`, `union(4, 9);` и `union(6, 5);`. Всякий раз, когда элемент (подмножество) становится ребенком другого элемента (подмножества), он модифицирует свое значение, чтобы отразить значение своего родителя (рис. 5.13).

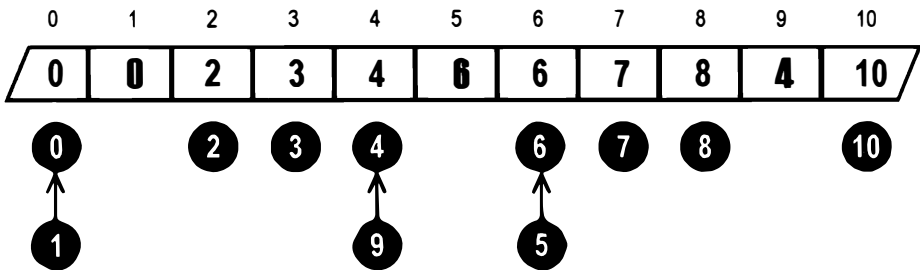


Рис. 5.13

Этот процесс продолжается до тех пор, пока мы не определим все наши подмножества. Например, мы можем добавить больше объединений — `union(0, 7)`, `union(4, 3)`, `union(4, 2)`, `union(6, 10)` и `union(4, 5)`. Это приведет к графическому представлению на рис. 5.14.

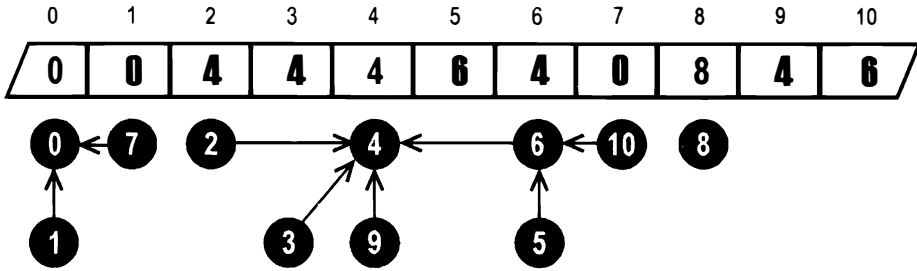


Рис 5.14

В качестве общего правила рекомендуется объединять меньшие подмножества с большими подмножествами, а не наоборот. Например, проверьте ситуацию, когда мы объединяем подмножество, содержащее 4, с подмножеством, содержащим 5. В этот момент 4 является родителем подмножества и имеет трех детей (2, 3 и 9), в то время как 5 находится рядом с 10, т. е. двумя детьми 6. Таким образом, подмножество, содержащее 5, имеет три узла (6, 5, 10), тогда как подмножество, содержащее 4, имеет четыре узла (4, 2, 3, 9). И поэтому 4 становится родителем 6 и, неявно, родителем 5.

В исходном коде эта работа касается массива `rank[]` (рис. 5.15).

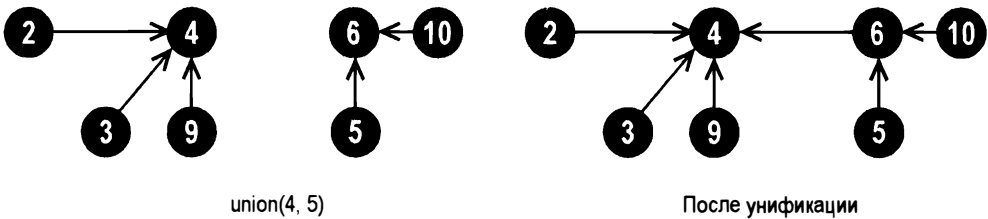


Рис 5.15

Давайте теперь посмотрим на то, как имплементировать операцию поиска и объединения.

Имплементирование операции поиска

Отыскание подмножества заданного элемента является рекурсивным процессом, который пересекает подмножество, следуя за родительскими элементами до тех пор, пока текущий элемент не станет родителем самого себя (корневым элементом):

```
public int find(int x) {
    if (parent[x] == x) {
        return x;
    } else {
        return find(parent[x]);
    }
}
```


Имплементирование операции объединения

Операция *объединения* начинается с извлечения корневых элементов из заданных подмножеств. Кроме того, если эти два корня различны, они должны опираться на свой ранг, чтобы решить, какой из них станет родителем другого (более высокий ранг становится родителем). Если они имеют одинаковый ранг, то следует выбрать один из них и увеличить его ранг на 1:

```
public void union(int x, int y) {
    int xRoot = find(x);
    int yRoot = find(y);

    if (xRoot == yRoot) {
        return;
    }

    if (rank[xRoot] < rank[yRoot]) {
        parent[xRoot] = yRoot;
    } else if (rank[yRoot] < rank[xRoot]) {
        parent[yRoot] = xRoot;
    } else {
        parent[yRoot] = xRoot;
        rank[xRoot]++;
    }
}
```

ОК. Давайте теперь определим дизъюнктивное множество:

```
DisjointSet set = new DisjointSet(11);
set.union(0, 1);
set.union(4, 9);
set.union(6, 5);
set.union(0, 7);
set.union(4, 3);
set.union(4, 2);
set.union(6, 10);
set.union(4, 5);
```

А теперь давайте поиграем с ним:

```
// 4 и 0 - друзья => false
System.out.println("4 и 0 являются друзьями: "
    + (set.find(0) == set.find(4)));

// 4 и 5 - друзья => true
System.out.println("4 и 5 являются друзьями: "
    + (set.find(4) == set.find(5)));
```

Этот алгоритм может быть оптимизирован путем сжатия путей между элементами. Например, взгляните на рис. 5.16.

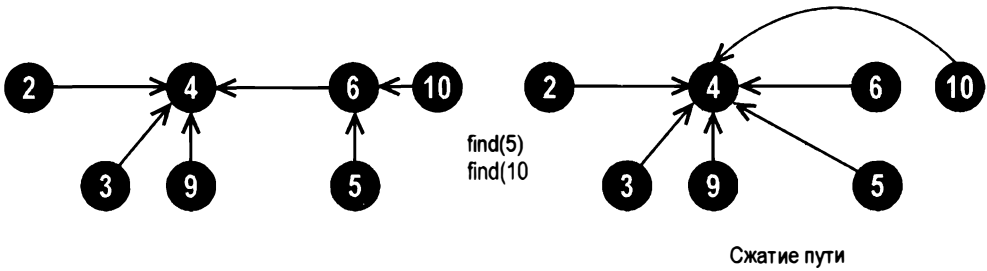


Рис. 5.16

С левой стороны, пытаясь найти родителя 5, вы должны пройти через 6, пока не достигнете 4. Точно так же, пытаясь найти родителя 10, вы должны пройти через 6, пока не достигнете 4. Однако с правой стороны мы сжимаем пути 5 и 10, связывая их непосредственно с 4. На этот раз мы можем найти родителя 5 и 10, не проходя через промежуточные элементы.

Сжатие пути может происходить по отношению к операции `find()` следующим образом:

```
public int find(int x) {
    if (parent[x] != x) {
        return parent[x] = find(parent[x]);
    }

    return parent[x];
}
```

Исходный код, прилагаемый к этой книге, содержит оба приложения как со сжатием пути, так и без него.

127. Дерево Фенвика, или двоичное индексированное дерево

Дерево Фенвика (Fenwick tree, FT), или двоичное индексированное дерево (binary indexed tree, BIT) — это массив, построенный для хранения сумм, соответствующих другому заданному массиву. Построенный массив имеет тот же размер, что и заданный массив, и каждая позиция (или узел) построенного массива хранит сумму некоторых элементов заданного массива. Поскольку Дерево Фенвика хранит частичные суммы заданного массива, оно является очень эффективным решением для вычисления суммы элементов заданного массива между двумя заданными индексами (диапазонных сумм/запросов) и исключает перебор значений между индексами в цикле и вычисление суммы.

Дерево Фенвика можно построить за линейное время или $O(n \log n)$. Очевидно, что мы предпочитаем делать это за линейное время, поэтому давайте посмотрим, как

мы можем это сделать. Начинаем с заданного (исходного) массива, который может иметь следующий вид (индексы представляют индекс в массиве):

$3_{(1)}, 1_{(2)}, 5_{(3)}, 8_{(4)}, 12_{(5)}, 9_{(6)}, 7_{(7)}, 13_{(8)}, 0_{(9)}, 3_{(10)}, 1_{(11)}, 4_{(12)}, 9_{(13)}, 0_{(14)}, 11_{(15)},$
 $5_{(16)}$

Идея построения дерева Фенвика основана на понятии наименьшего значащего бита (least significant bit, LSB). Точнее, допустим, что в данный момент мы имеем дело с элементом из индекса a . Тогда значение непосредственно над нами должно находиться в индексе b , где $b = a + \text{LSB}(a)$. Для того чтобы мы смогли применить алгоритм, значение из индекса 0 должно быть равно 0; поэтому массив, которым мы оперируем, выглядит следующим образом:

$0_{(0)}, 3_{(1)}, 1_{(2)}, 5_{(3)}, 8_{(4)}, 12_{(5)}, 9_{(6)}, 7_{(7)}, 13_{(8)}, 0_{(9)}, 3_{(10)}, 1_{(11)}, 4_{(12)}, 9_{(13)}, 0_{(14)},$
 $11_{(15)}, 5_{(16)}$

Теперь применим несколько шагов алгоритма и заполним дерево Фенвика суммами. В индексе 0 в дереве Фенвика мы имеем 0. Далее мы используем формулу $b = a + \text{LSB}(a)$ для вычисления оставшихся сумм следующим образом:

1. $a = 1$. Если $a = 1 = 00001_2$, то $b = 00001_2 + 00001_2 = 1 + 1 = 2 = 00010_2$. Мы говорим, что 2 отвечает за a (т. е. 1). Поэтому в дереве Фенвика с индексом 1 мы храним значение 3, а с индексом 2 — сумму значений $3 + 1 = 4$.
2. $a = 2$. Если $a = 2 = 00010_2$, то $b = 00010_2 + 00010_2 = 2 + 2 = 4 = 00100_2$. Мы говорим, что 4 отвечает за a (т. е. 2). Поэтому в дереве Фенвика с индексом 4 мы храним сумму значений $8 + 4 = 12$.
3. $a = 3$. Если $a = 3 = 00011_2$, то $b = 00011_2 + 00001_2 = 3 + 1 = 4 = 00100_2$. Мы говорим, что 4 отвечает за a (т. е. 3). Поэтому в дереве Фенвика с индексом 4 мы храним сумму значений $12 + 5 = 17$.
4. $a = 4$. Если $a = 4 = 00100_2$, то $b = 00100_2 + 00100_2 = 4 + 4 = 8 = 01000_2$. Мы говорим, что 8 отвечает за a (т. е. 4). Поэтому в дереве Фенвика с индексом 8 мы храним сумму значений $13 + 17 = 30$.

Данный алгоритм будет продолжаться в таком же стиле до тех пор, пока дерево Фенвика не будет завершено. В графическом представлении наш случай может быть сформирован так, как показано на рис. 5.17.



Если вычисленная точка индекса находится за пределами границ, то следует ее просто проигнорировать.

Значение	Индекс	Двоичное число	LSB	Диапазон ответственности	BIT
5	16	10000 ₂	2 ⁴		91
11	15	01111 ₂	2 ⁰	1	11
0	14	01110 ₂	2 ¹	2	9
9	13	01101 ₂	2 ¹	1	9
4	12	01100 ₂	2 ²		8
1	11	01011 ₂	2 ⁰	1	1
3	10	01010 ₂	2 ¹	4	3
0	9	01001 ₂	2 ⁰	2	
13	8	01000 ₂	2 ³	16	0
7	7	00111 ₂	2 ⁰	1	7
9	6	00110 ₂	2 ¹		21
12	5	00101 ₂	2 ⁰	2	12
8	4	00100 ₂	2 ²	8	17
5	3	00011 ₂	2 ⁰	1	5
1	2	00010 ₂	2 ¹	4	4
3	1	00001 ₂	2 ⁰	2	3
0	0	-		1	0

Рис. 5.17

В исходном коде приведенный выше поток может быть сформирован следующим образом (значения — это заданный массив):

```
public class FenwickTree {
    private final int n;
    private long[] tree;
    ...

    public FenwickTree(long[] values) {

        values[0] = 0 L;
        this.n = values.length;
        tree = values.clone();

        for (int i = 1; i < n; i++) {

            int parent = i + lsb(i);
            if (parent < n) {
                tree[parent] += tree[i];
            }
        }
    }
}
```

```
private static int lsb(int i) {
    return i & -i;

    // либо
    // return Integer.lowestOneBit(i);
}

...
}
```

Теперь дерево Фенвика (BIT) готово, и мы можем выполнять обновления и диапазонные запросы.

Например, для того чтобы выполнить диапазонное суммирование, мы должны получить соответствующие диапазоны и их просуммировать. Рассмотрим несколько примеров в правой части схемы на рис. 5.18, чтобы быстро понять этот процесс.

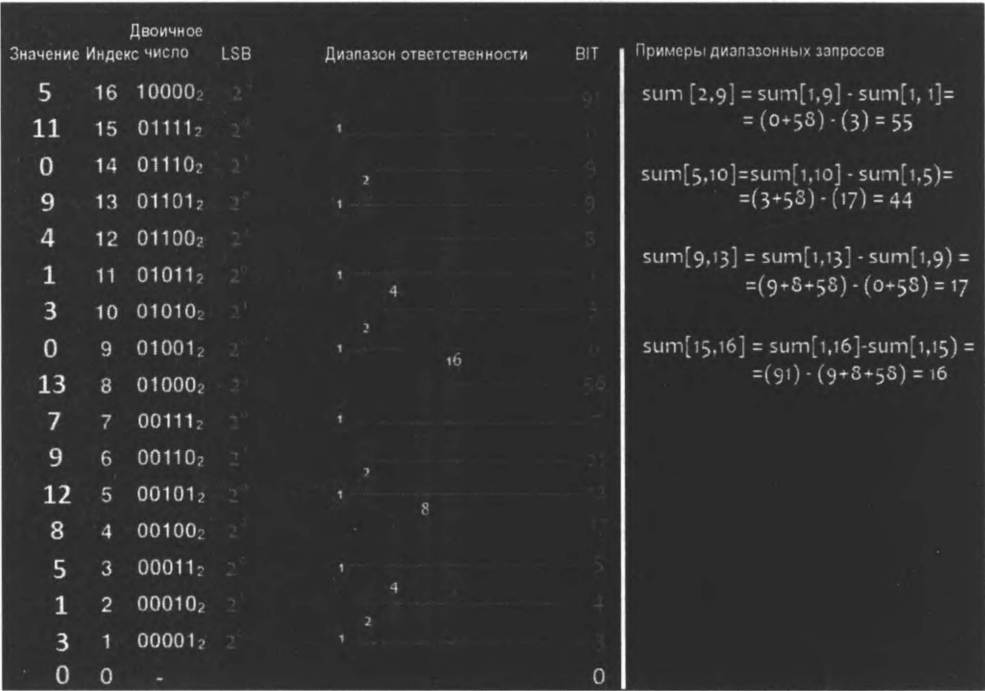


Рис. 5.18

С точки зрения исходного кода, это может быть легко сформировано следующим образом:

```
public long sum(int left, int right) {
    return prefixSum(right) - prefixSum(left - 1);
}
```

```
private long prefixSum(int i) {
    long sum = 0L;

    while (i != 0) {
        sum += tree[i];
        i &= ~lsb(i); // либо i -= lsb(i);
    }

    return sum;
}
```

Кроме того, мы можем добавить новое значение:

```
public void add(int i, long v) {
    while (i < n) {
        tree[i] += v;
        i += lsb(i);
    }
}
```

И мы также можем установить новое значение, равным конкретному индексу:

```
public void set(int i, long v) {
    add(i, v - sum(i, i));
}
```

Имея все эти функции на одном месте, мы можем создать дерево Фенвика для нашего массива следующим образом:

```
FenwickTree tree = new FenwickTree(new long[] {
    0, 3, 1, 5, 8, 12, 9, 7, 13, 0, 3, 1, 4, 9, 0, 11, 5
});
```

И тогда мы сможем с ним поиграть:

```
long sum29 = tree.sum(2, 9); // 55
tree.set(4, 3);
tree.add(4, 5);
```

128. Фильтр Блума

Фильтр Блума — это быстрая и эффективная для памяти структура данных, способная давать вероятностный ответ на вопрос *о принадлежности значения X заданному множеству?*

Обычно этот алгоритм полезен, когда множество имеет огромный размер, и большинство алгоритмов поиска сталкиваются с проблемами памяти и скорости.

Скорость и эффективность работы фильтра Блума обусловлены тем, что эта структура данных основана на массиве битов (например, `ava.util.BitSet`). Изначально биты этого массива имеют значение 0 или `false`.

Битовый массив является первым главным ингредиентом фильтра Блума. Второй главный компонент состоит из одной или нескольких хеш-функций. В идеале эти

хеш-функции являются *попарно независимыми и равномерно распределенными*. Кроме того, очень важно быть чрезвычайно быстрым. Хеш-функции Murmur, серия fnv и HashMix — вот лишь несколько хеш-функций, которые соблюдают этим ограничением в приемлемой степени для использования фильтром Блума.

Теперь, когда мы добавляем элемент в фильтр Блума, нам нужно прохешировать этот элемент (пропустить его через каждую имеющуюся хеш-функцию) и установить биты в битовом массиве с индексом этих хешей, равным 1 или true.

Следующий фрагмент кода должен прояснить главную идею:

```
private BitSet bitset; // битовый массив
private static final Charset CHARSET = StandardCharsets.UTF_8;
...

public void add(T element) {
    add(element.toString().getBytes(CHARSET));
}

public void add(byte[] bytes) {
    int[] hashes = hash(bytes, numberOfHashFunctions);

    for (int hash: hashes) {
        bitset.set(Math.abs(hash % bitSetSize), true);
    }

    numberOfAddedElements++;
}
```

Теперь, когда мы ищем элемент, мы пропускаем этот элемент через те же хеш-функции. Кроме того, мы проверяем, что результирующие значения в битовом массиве помечены 1 или true. Если они не помечены, то элемент точно не принадлежит множеству. Если же они помечены, то мы знаем с некоторой вероятностью, что элемент принадлежит множеству. Это не 100%-ная точность, т. к. другой элемент или комбинация элементов могли перевернуть эти биты. Неправильные ответы называются *ложными утверждениями (ложными срабатываниями)*.

В терминах исходного кода мы имеем следующее:

```
private BitSet bitset; // битовый массив
private static final Charset CHARSET = StandardCharsets.UTF_8;
...

public boolean contains(T element) {
    return contains(element.toString().getBytes(CHARSET));
}

public boolean contains(byte[] bytes) {
    int[] hashes = hash(bytes, numberOfHashFunctions);
```

```

for (int hash: hashes) {
    if (!bitset.get(Math.abs(hash % bitSetSize))) {
        return false;
    }
}

return true;
}

```

В графическом виде мы можем представить фильтр Блума битовым массивом размером 11 и тремя хеш-функциями так, как показано на рис. 5.19 (мы добавили два элемента).

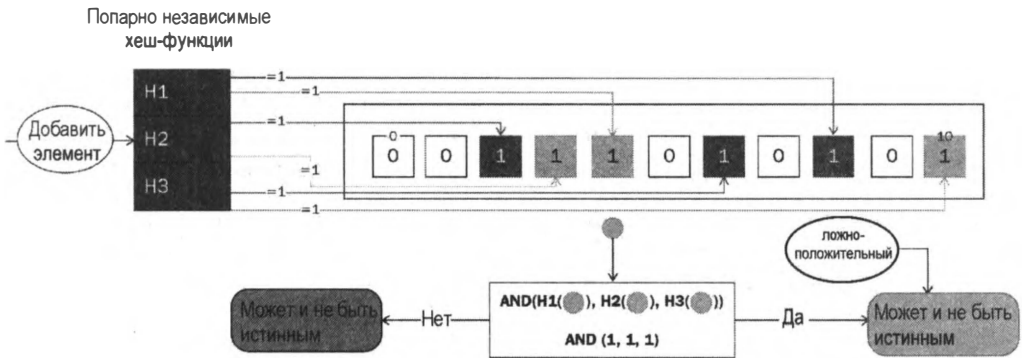


Рис. 5.19

Очевидно, что мы хотим максимально сократить число ложных утверждений. Хотя мы не можем полностью устранить их, мы все же можем повлиять на их частоту, регулируя размер битового массива, число хеш-функций и число элементов в множестве.

Для формирования оптимального фильтра Блума можно использовать следующие математические формулы:

- ◆ число элементов в фильтре (может быть оценено на основе m , k и p):

$$n = \text{ceil}(m / (-k / \log(1 - \exp(\log(p) / k))));$$
- ◆ вероятность ложных утверждений, дробь между 0 и 1 или число, указывающее на 1 из p :

$$p = \text{pow}(1 - \exp(-k / (m / n)), k);$$
- ◆ число битов в фильтре (или размер в терминах килобайтов, мегабайтов, гигабайтов и т. д.):

$$m = \text{ceil}((n * \log(p)) / \log(1 / \text{pow}(2, \log(2))));$$
- ◆ число хеш-функций (может быть оценено на основе m и n):

$$k = \text{round}((m / n) * \log(2));$$



В качестве общего правила следует принять, что больший фильтр будет иметь меньше ложных утверждений, чем меньший. Кроме того, увеличивая число хеш-функций, мы получаем меньше ложных утверждений, но при этом замедляем работу фильтра и быстро его заполняем. Производительность фильтра Блума равна $O(h)$, где h — это число хеш-функций.

В исходном коде, прилагаемом к этой книге, имеется имплементация фильтра Блума с использованием хеш-функций на основе SHA-256 и murmur. Поскольку этот код слишком длинный, чтобы уместиться в этой книге, рассмотрите самостоятельно в качестве отправной точки пример из класса `Main`.

Резюме

В этой главе было рассмотрено 30 задач с привлечением массивов, коллекций и нескольких структур данных. Задачи, охватывающие массивы и коллекции, являются частью повседневной работы, а в задачах, охватывающих структуры данных, мы ввели несколько менее известных (но мощных) структур данных, таких как дерево Фенвика, дизъюнктивные множества Union Find и префиксное дерево.

Вы можете скачать приложения из этой главы, чтобы увидеть результаты и познакомиться с дополнительными деталями, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

6

Пути, файлы, буферы, сканирование и форматирование ввода/вывода в среде Java

Эта глава содержит 20 задач с привлечением файлового ввода/вывода в среде Java. Мы рассмотрим задачи, с которыми разработчики в среде Java сталкиваются ежедневно, от манипулирования путями, их посещения и наблюдения за ними до потоковой передачи файлов и эффективных способов чтения/записи текстовых и двоичных файлов.

Благодаря навыкам, полученным из этой главы, вы сможете решать большинство часто встречающихся задач, связанных с файловым вводом/выводом в среде Java. Широкий диапазон тем в этой главе предоставит массу информации о том, как Java выполняет операции ввода/вывода.

Задачи

Используйте следующие задачи для проверки вашего умения программировать ввод/вывод в среде Java. Перед тем как обратиться к решениям и скачать примеры программ, настоятельно рекомендуется изучить каждую задачу.

129. **Создание путей к файлам.** Написать несколько примеров создания разных типов путей к файлам (например, абсолютные пути, относительные пути и т. д.).
130. **Конвертирование путей к файлам.** Написать несколько примеров конвертирования путей к файлам (например, конвертирование пути к файлу в строку, URI-идентификатор, файл и т. д.).
131. **Присоединение путей к именам файлов.** Написать несколько примеров присоединения (совмещения) путей к файлам. Определить фиксированный путь и добавить к нему другие пути (или заменить его часть другими путями).

132. **Конструирование пути между двумя местоположениями.** Написать несколько примеров, которые конструируют относительный путь между двумя заданными путями (от одного пути к другому).
133. **Сравнение путей к файлам.** Написать несколько примеров сравнения указанных путей к файлам.
134. **Посещение путей.** Написать программу, которая посещает все файлы в каталоге, включая подкаталоги. Кроме того, написать программу, которая ищет файл по имени, удаляет каталог, перемещает каталог и копирует каталог.
135. **Наблюдение за путями.** Написать несколько программ, которые наблюдают за изменениями, происходящими на определенном пути (например, создание, удаление и изменение).
136. **Потоковая передача содержимого файла.** Написать программу, которая будет передавать содержимое заданного файла в потоке.
137. **Поиск файлов/папок в дереве файлов.** Написать программу, которая ищет заданные файлы/папки в заданном дереве файлов.
138. **Эффективное чтение/запись текстовых файлов.** Написать несколько программ, чтобы продемонстрировать разные подходы к эффективному чтению и записи текстового файла.
139. **Эффективное чтение/запись двоичных файлов.** Написать несколько программ, чтобы продемонстрировать разные подходы к эффективному чтению и записи двоичного файла.
140. **Поиск в больших файлах.** Написать программу, которая эффективно отыскивает заданное строковое значение в большом файле.
141. **Чтение файла JSON/CSV как объекта.** Написать программу, которая читает заданный файл JSON/CSV как обыкновенный объект Java (POJO).
142. **Работа с временными файлами/папками.** Написать несколько программ для работы с временными файлами/папками.
143. **Фильтрация файлов.** Написать несколько пользовательских фильтров для файлов.
144. **Обнаружение несовпадений между двумя файлами.** Написать программу, которая обнаруживает несовпадения между двумя файлами на байтовом уровне.
145. **Кольцевой байтовый буфер.** Написать программу, представляющую имплементацию кольцевого байтового буфера.
146. **Лексемизация файлов.** Написать несколько фрагментов кода, чтобы проиллюстрировать различные методы разбора содержимого файла на лексемы (токены).

147. **Запись форматированного вывода непосредственно в файл.** Написать программу, которая форматирует заданные числа (целые числа и числа двойной точности) и выводит их в файл.
148. **Работа с классом `Scanner`.** Написать несколько фрагментов кода, для того чтобы раскрыть возможности класса `Scanner`.

Решения

В следующих разделах описаны решения перечисленных выше задач. Помните, что обычно не существует одного единственно правильного варианта решения той или иной задачи. Кроме того, имейте в виду, что приведенные здесь объяснения включают только самые интересные и важные детали, необходимые для решения. Вы можете скачать примеры решений, чтобы увидеть дополнительные детали и поэкспериментировать с программами, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

129. Создание путей к файлам

Начиная с JDK 7, мы можем создавать путь к файлу посредством API неблокирующего ввода/вывода NIO.2. Точнее, путь к файлу можно легко определять с помощью API `Path` и `Paths`.

Класс `Path` — это программное представление пути в файловой системе. Строка пути содержит следующую информацию:

- ◆ имя файла;
- ◆ список каталогов;
- ◆ зависимый от операционной системы разделитель файлов (например, прямая косая черта `/` в Solaris и Linux и обратная косая черта `\` в Microsoft Windows);
- ◆ другие разрешенные символы, например обозначения `.` (текущий каталог) и `..` (родительский каталог).



Класс `Path` работает с файлами в разных файловых системах (`FileSystem`), который может использовать разные места хранения (`FileStore` является базовым хранилищем).

Часто встречающимся вариантом решения задачи определения пути является вызов одной из разновидностей метода `get()` вспомогательного класса `Paths`. Еще один вариант решения опирается на метод `FileSystems.getDefault().getPath()`.

Объект `Path` располагается в файловой системе — *файловая система хранит и организует файлы или некоторые формы носителей информации, обычно на одном или нескольких жестких дисках таким образом, чтобы их можно было легко извлечь*. Файловую систему можно получить через финальный класс `java.nio.file.FileSystems`,

который используется для получения экземпляра класса `java.nio.file.FileSystem`. Экземпляр `FileSystem` по умолчанию JVM-машины (обычно именуемый файловой системой по умолчанию операционной системы) может быть получен посредством метода `FileSystems().getDefault()`. Узнав файловую систему и расположение файла (или каталога/папки), мы можем создать для нее объект `Path`.

Еще один подход заключается в создании пути из унифицированного идентификатора ресурса (URI). Java обертывает URI посредством класса `URI`; затем мы можем получить URI-идентификатор из значения типа `String` посредством метода `URI.create(String uri)`. Кроме того, класс `Paths` предоставляет метод `get()`, который принимает объект `URI` в качестве аргумента и возвращает соответствующий объект `Path`.

Начиная с JDK 11, мы можем создавать путь посредством двух методов `of()`. Один из них конвертирует URI в `Path`, тогда как другой конвертирует строку пути или последовательность строк, соединенных в строку пути.

В следующих далее разделах мы рассмотрим различные способы создания путей.

Создание пути относительно корня хранилища файлов

Путь, расположенный относительно текущего корня хранилища файлов (например, `C:/`), должен начинаться с разделителя файлов. В приведенных далее примерах если текущим корнем хранилища файлов является `C`, то абсолютный путь равен `C:\learning\packt\JavaModernChallenge.pdf`:

```
Path path = Paths.get("/learning/packt/JavaModernChallenge.pdf");
Path path = Paths.get("/learning", "packt/JavaModernChallenge.pdf");
```

```
Path path = Path.of("/learning/packt/JavaModernChallenge.pdf");
Path path = Path.of("/learning", "packt/JavaModernChallenge.pdf");
```

```
Path path = FileSystems.getDefault()
    .getPath("/learning/packt", "JavaModernChallenge.pdf");
Path path = FileSystems.getDefault()
    .getPath("/learning/packt/JavaModernChallenge.pdf");
```

```
Path path = Paths.get(
    URI.create("file:///learning/packt/JavaModernChallenge.pdf"));
Path path = Path.of(
    URI.create("file:///learning/packt/JavaModernChallenge.pdf"));
```

Создание пути относительно текущей папки

Когда мы создаем путь, который находится относительно текущей рабочей папки, этот путь *не* должен начинаться с разделителя файлов. Если текущая папка называется

ся books и находится под корнем C, тогда абсолютный путь, возвращаемый следующим ниже фрагментом кода, будет C:\books\learning\packt\JavaModernChallenge.pdf:

```
Path path = Paths.get("learning/packt/JavaModernChallenge.pdf");
Path path = Paths.get("learning", "packt/JavaModernChallenge.pdf");

Path path = Path.of("learning/packt/JavaModernChallenge.pdf");
Path path = Path.of("learning", "packt/JavaModernChallenge.pdf");

Path path = FileSystems.getDefault()
    .getPath("learning/packt", "JavaModernChallenge.pdf");
Path path = FileSystems.getDefault()
    .getPath("learning/packt/JavaModernChallenge.pdf");
```

Создание абсолютного пути

Создание абсолютного пути выполняется путем явного указания корневого каталога и всех других подкаталогов, содержащих файл или папку, как показано в следующих ниже примерах (C:\learning\packt\JavaModernChallenge.pdf):

```
Path path = Paths.get("C:/learning/packt", "JavaModernChallenge.pdf");
Path path = Paths.get("C:", "learning/packt", "JavaModernChallenge.pdf");
Path path = Paths.get("C:", "learning", "packt", "JavaModernChallenge.pdf");
Path path = Paths.get("C:/learning/packt/JavaModernChallenge.pdf");
Path path = Paths.get(
    System.getProperty("user.home"), "downloads", "chess.exe");

Path path = Path.of("C:", "learning/packt", "JavaModernChallenge.pdf");
Path path = Path.of(System.getProperty("user.home"), "downloads", "chess.exe");

Path path = Paths.get(URI.create(
    "file:///C:/learning/packt/JavaModernChallenge.pdf"));
Path path = Path.of(URI.create(
    "file:///C:/learning/packt/JavaModernChallenge.pdf"));
```

Создание пути с помощью сокращений

Под сокращениями мы понимаем обозначения . (текущий каталог) и .. (родительский каталог). Этот тип пути может быть нормализован посредством метода `normalize()`. Указанный метод устраняет избыточность, такую как . и `directory/..`:

```
Path path = Paths.get(
    "C:/learning/packt/chapters/../JavaModernChallenge.pdf")
    .normalize();
Path path = Paths.get(
    "C:/learning/./packt/chapters/../JavaModernChallenge.pdf")
    .normalize();
```

```
Path path = FileSystems.getDefault()
    .getPath("/learning/./packt", "JavaModernChallenge.pdf")
    .normalize();
```

```
Path path = Path.of(
    "C:/learning/packt/chapters/./JavaModernChallenge.pdf")
    .normalize();
```

```
Path path = Path.of(
    "C:/learning/./packt/chapters/./JavaModernChallenge.pdf")
    .normalize();
```



Без нормализации избыточные части пути не удаляются.

В создании путей, которые на 100% совместимы с текущей операционной системой, мы можем опереться на метод `FileSystems.getDefault().getPath()` либо на комбинацию разделителя `File.separator` (зависящего от системы дефолтного символа разделителя имен) и метода `File.listRoots()` (возвращающего имеющиеся корни файловой системы). В случае относительных путей мы можем воспользоваться такими примерами:

```
private static final String FILE_SEPARATOR = File.separator;
```

В качестве альтернативы мы можем вызвать метод `getSeparator()`:

```
private static final String FILE_SEPARATOR
    = FileSystems.getDefault().getSeparator();
```

```
// относительно текущей рабочей папки
```

```
Path path = Paths.get("learning", "packt", "JavaModernChallenge.pdf");
Path path = Path.of("learning", "packt", "JavaModernChallenge.pdf");
Path path = Paths.get(String.join(FILE_SEPARATOR, "learning",
    "packt", "JavaModernChallenge.pdf"));
Path path = Path.of(String.join(FILE_SEPARATOR, "learning",
    "packt", "JavaModernChallenge.pdf"));
```

```
// относительно корня хранилища файлов
```

```
Path path = Paths.get(FILE_SEPARATOR + "learning",
    "packt", "JavaModernChallenge.pdf");
Path path = Path.of(FILE_SEPARATOR + "learning",
    "packt", "JavaModernChallenge.pdf");
```

То же самое можно сделать и для абсолютных путей:

```
Path path = Paths.get(File.listRoots()[0] + "learning",
    "packt", "JavaModernChallenge.pdf");
Path path = Path.of(File.listRoots()[0] + "learning",
    "packt", "JavaModernChallenge.pdf");
```

Список корневых каталогов также можно получить посредством класса `FileSystems`:

```
FileSystems.getDefault().getRootDirectories()
```

130. Конвертирование путей к файлам

Операция конвертирования пути к файлу в объекты `String`, `URI`, `File` и подобные им встречается часто и может выполняться в широком диапазоне приложений. Рассмотрим следующий путь к файлу:

```
Path path = Paths.get("/learning/packt", "JavaModernChallenge.pdf");
```

Теперь на основе `JDK 7` и `API` неблокирующего ввода/вывода `NIO.2` посмотрим, каким образом можно конвертировать объект `Path` в объект `String`, `URI`, абсолютный путь, *реальный* путь и объект `File`.

- ◆ Конвертировать объект `Path` в объект `String` так же просто, как вызвать (явно или автоматически) метод `Path.toString()`. Обратите внимание, что если путь был получен посредством метода `FileSystem.getPath()`, то строка пути, возвращаемая методом `toString()`, может отличаться от исходной строки, которая использовалась для создания пути:

```
// \learning\packt\JavaModernChallenge.pdf
String pathToString = path.toString();
```

- ◆ Конвертировать объект `Path` в объект `URI` (в браузерном формате) можно посредством метода `Path.toURI()`. Возвращаемый объект `URI` обертывает строку пути, и его можно использовать в адресной строке веб-браузера:

```
// file:///D:/learning/packt/JavaModernChallenge.pdf
URI pathToURI = path.toUri();
```

Предположим, что мы хотим извлечь имя файла, присутствующее в `URI/URL` в качестве пути (такой сценарий очень распространен). В подобных случаях мы можем опереться на приведенные далее фрагменты кода:

```
// JavaModernChallenge.pdf
URI uri = URI.create(
    "https://www.learning.com/packt/JavaModernChallenge.pdf");
Path URIToPath = Paths.get(uri.getPath()).getFileName();

// JavaModernChallenge.pdf
URL url = new URL(
    "https://www.learning.com/packt/JavaModernChallenge.pdf");
Path URLToPath = Paths.get(url.getPath()).getFileName();
```

Конвертировать пути можно следующим образом.

- ◆ Конвертировать относительный путь в абсолютный можно посредством метода `Path.toAbsolutePath()`. Если путь уже является абсолютным, то будет возвращен тот же результат:

```
// D:\learning\packt\JavaModernChallenge.pdf
Path pathToAbsolutePath = path.toAbsolutePath();
```


- ◆ Конвертировать объект `Path` в реальный путь можно посредством метода `Path.toRealPath()`, и его результат зависит от реализации. Если указываемый файл не существует, этот метод выбросит исключение `IOException`. Но, как правило, результатом вызова этого метода является абсолютный путь без избыточных элементов (нормализованный). Этот метод получает аргумент, который указывает на то, как нужно трактовать символические ссылки. По умолчанию, если файловая система поддерживает символические ссылки, этот метод попытается их определить и обработать. Если вы хотите игнорировать символические ссылки, то просто передайте в метод константу `LinkOption.NOFOLLOW_LINKS`. Кроме того, элементы имени пути будут представлять фактическое имя каталогов и файла.

Например, рассмотрим следующий ниже путь и результат вызова этого метода (обратите внимание, что мы намеренно добавили несколько избыточных элементов и написали папку `PACKT` заглавными буквами):

```
Path path = Paths.get(
    "/learning/books/../../PACKT/./", "JavaModernChallenge.pdf");

// D:\learning\packt\JavaModernChallenge.pdf
Path realPath = path.toRealPath(LinkOption.NOFOLLOW_LINKS);
```

- ◆ Конвертировать объект `Path` в объект `File` можно посредством метода `PathToFile()`. В конвертировании объекта `File` в объект `Path` мы можем опереться на метод `File.toPath()`:

```
File pathToFile = path.toFile();
Path fileToPath = pathToFile.toPath();
```

131. Присоединение путей к именам файлов

Соединение (или сочетание) путей файлов означает определение фиксированного корневого пути и добавление к нему частичного пути или замену его части (например, имя файла должно быть заменено другим именем файла). В принципе, этот технический прием удобен, когда мы хотим создать новые пути, которые имеют общую фиксированную часть.

Это можно достичь посредством API неблокирующего ввода/вывода NIO.2 и методов `Path.resolve()` и `Path.resolveSibling()`.

Рассмотрим следующий фиксированный корневой путь:

```
Path base = Paths.get("D:/learning/packt");
```

Давайте также допустим, что мы хотим получить путь для двух разных книг:

```
// D:\learning\packt\JBossTools3.pdf
Path path = base.resolve("JBossTools3.pdf");
```

```
// D:\learning\packt\MasteringJSF22.pdf
Path path = base.resolve("MasteringJSF22.pdf");
```

Мы можем использовать эту функцию для перебора множества файлов в цикле; например, давайте выполним обход массива типа `String[]` книг:

```
Path basePath = Paths.get("D:/learning/packt");
String[] books = {
    "Book1.pdf", "Book2.pdf", "Book3.pdf"
};

for (String book: books) {
    Path nextBook = basePath.resolve(book);
    System.out.println(nextBook);
}
```

Иногда фиксированный корневой путь также содержит имя файла:

```
Path base = Paths.get("D:/learning/packt/JavaModernChallenge.pdf");
```

На этот раз мы можем заменить имя файла (`JavaModernChallenge.pdf`) другим именем посредством метода `resolveSibling()`. Этот метод регулирует заданный путь относительно родительского пути этого пути, как показано в следующем примере:

```
// D:\learning\packt\MasteringJSF22.pdf
Path path = base.resolveSibling("MasteringJSF22.pdf");
```

Если же взять метод `Path.getParent()` и выстроить цепочку из методов `resolve()` и `resolveSibling()`, мы можем создать более сложные пути:

```
// D:\learning\publisher\MyBook.pdf
Path path = base.getParent().resolveSibling("publisher")
    .resolve("MyBook.pdf");
```

Методы `resolve()/resolveSibling()` идут в двух разновидностях: соответственно `resolve(String other)/resolveSibling(String other)` и `resolve(Path other)/resolveSibling(Path other)`.

132. Конструирование пути между двумя местоположениями

Конструированием относительного пути между двумя местоположениями занимается метод `Path.relativeize()`.

Результирующий относительный путь (возвращаемый из `Path.relativeize()`) начинается с пути и заканчивается на другом пути. Это средство является довольно мощным и позволяет перемещаться между разными местоположениями, используя относительные пути, которые регулируются относительно предыдущих путей.

Рассмотрим следующие ниже два пути:

```
Path path1 = Paths.get("JBossTools3.pdf");
Path path2 = Paths.get("JavaModernChallenge.pdf");
```

Обратите внимание, что файлы `JBossTools3.pdf` и `JavaModernChallenge.pdf` являются одноуровневыми. Это означает, что мы можем перемещаться от одного к другому,

поднимаясь на один уровень вверх, а затем спускаясь на один уровень вниз. Этот случай навигации проявляется также на следующих примерах:

```
// ..\JavaModernChallenge.pdf
Path path1ToPath2 = path1.relativize(path2);
```

```
// ..\JBossTools3.pdf
Path path2ToPath1 = path2.relativize(path1);
```

Еще один часто встречающийся случай связан с общим корневым элементом:

```
Path path3 = Paths.get("/learning/packt/2003/JBossTools3.pdf");
Path path4 = Paths.get("/learning/packt/2019");
```

Таким образом, `path3` и `path4` имеют одинаковый общий корневой элемент `/learning`. Для перехода от `path3` к `path4` нам нужно подняться на два уровня вверх и спуститься на один уровень вниз. В дополнение к этому, для перехода от `path4` к `path3` нам нужно подняться на один уровень вверх и спуститься на два уровня вниз. Взгляните на следующий фрагмент кода:

```
// ..\..\2019
Path path3ToPath4 = path3.relativize(path4);

// ..\2003\JBossTools3.pdf
Path path4ToPath3 = path4.relativize(path3);
```



Оба пути должны содержать корневой элемент. Выполнение этого требования не гарантирует успеха, поскольку конструирование относительного пути зависит от имплементации.

133. Сравнение путей к файлам

Существует несколько вариантов решения этой задачи в зависимости от того, как мы воспринимаем эквивалентность между двумя путями к файлам. В общих чертах эквивалентность путей может быть проверена разными способами для разных целей.

Допустим, что у нас есть три пути (рассмотрим воспроизведение `path3` на вашем компьютере):

```
Path path1 = Paths.get("/learning/packt/JavaModernChallenge.pdf");
Path path2 = Paths.get("/LEARNING/PACKT/JavaModernChallenge.pdf");
Path path3 = Paths.get("D:/learning/packt/JavaModernChallenge.pdf");
```

В следующих разделах мы рассмотрим разные методы, используемые для сравнения путей к файлам.

Метод *Path.equals()*

Эквивалентны ли `path1` и `path2`? Или же `path2` эквивалентен `path3`? Так вот, если мы выполним эти проверки посредством метода `Path.equals()`, то возможный результат покажет, что `path1` эквивалентен `path2`, но `path2` не эквивалентен `path3`:

```
boolean path1EqualsPath2 = path1.equals(path2); // true
boolean path2EqualsPath3 = path2.equals(path3); // false
```

Метод `Path.equals()` подчиняется спецификации метода `Object.equals()`. Хотя этот метод не обращается к файловой системе, эквивалентность зависит от реализации файловой системы. Например, некоторые реализации файловой системы сравнивают пути с учетом регистра, в то время как другие регистр игнорируют.

Пути, представляющие одинаковый файл/папку

Однако это, вероятно, не то сравнение, которое нам нужно. Разумнее сказать, что два пути эквивалентны, если они являются одним и тем же файлом или папкой. Это можно сделать посредством метода `Files.isSameFile()`, который действует в два этапа:

Во-первых, он вызывает `Path.equals()`, и если этот метод возвращает `true`, то пути являются эквивалентными и не требуют дальнейших действий.

Во-вторых, если `Path.equals()` возвращает `false`, то он проверяет, представляют ли оба пути одинаковый файл/папку (в зависимости от реализации это действие может потребоваться для открытия/доступа к обоим файлам, поэтому файлы должны существовать во избежание исключения `IOException`).

```
//true
boolean path1IsSameFilePath2 = Files.isSameFile(path1, path2);
//true
boolean path1IsSameFilePath3 = Files.isSameFile(path1, path3);
//true
boolean path2IsSameFilePath3 = Files.isSameFile(path2, path3);
```

Лексикографическое сравнение

Если вам нужно лишь сравнить пути лексикографически, то мы можем опереться на метод `Path.compareTo()` (он бывает полезен для сортировки).

Этот метод возвращает следующую информацию:

- ◆ 0, если пути являются эквивалентными;
- ◆ значение меньше нуля, если первый путь лексикографически меньше пути аргумента;
- ◆ значение больше нуля, если первый путь лексикографически больше пути аргумента:

```
int path1compareToPath2 = path1.compareTo(path2);    // 0
int path1compareToPath3 = path1.compareTo(path3);    // 24
int path2compareToPath3 = path2.compareTo(path3);    // 24
```

Обратите внимание, что вы можете получить другие значения, нежели в приведенном выше примере. Далее в вашей бизнес-логике важно опереться на их смысл, а не на их значение (например, `if(path1compareToPath3 > 0) { ... }`, и следует избегать `if(path1compareToPath3 == 24) { ... }`).

Частичное сравнение

Частичное сравнение достижимо посредством методов `Path.startsWith()` и `Path.endsWith()`. Используя эти методы, мы можем проверить, что текущий путь начинается/заканчивается заданным путем:

```
boolean sw = path1.startsWith("/learning/packt"); // true
boolean ew = path1.endsWith("JavaModernChallenge.pdf"); // true
```

134. Прохождение путей

Существуют разные варианты решения задачи прохождения путей (или их посещения), и одно из них предоставляется API неблокирующего ввода/вывода NIO.2 посредством интерфейса `FileVisitor`.

Этот интерфейс выставляет наружу набор методов, представляющих контрольные точки в рекурсивном процессе посещения заданного пути. Игнорируя эти контрольные точки, мы имеем право вмешиваться в этот процесс. Мы можем обработать посещаемый файл/папку и принимать решение о том, что должно происходить дальше посредством специального объекта — перечисления `FileVisitResult` (как подкласса класса перечислений `Enum`), которое содержит следующие константы:

- ◆ `CONTINUE` — процесс обхода должен продолжиться (посетить следующий файл, папку, пропустить сбой и т. д.);
- ◆ `SKIP_SIBLINGS` — процесс обхода должен продолжиться без посещения одноуровневых элементов текущего файла/папки;
- ◆ `SKIP_SUBTREE` — процесс обхода должен продолжиться без посещения элементов в текущей папке;
- ◆ `TERMINATE` — обход должен быть жестоко прерван.

Интерфейс `FileVisitor` выставляет наружу следующие методы:

- ◆ `FileVisitResult visitFile(T file, BasicFileAttributes attrs) throws IOException` — автоматически вызывается для каждого посещенного файла/папки;
- ◆ `FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs) throws IOException` — автоматически вызывается для папки перед посещением ее содержимого;
- ◆ `FileVisitResult postVisitDirectory(T dir, IOException exc) throws IOException` — автоматически вызывается после посещения содержимого в каталоге (включая потомков) либо во время итерации по папке произошла ошибка ввода/вывода или посещение было прервано программно;
- ◆ `FileVisitResult visitFileFailed(T file, IOException exc) throws IOException` — автоматически вызывается, когда не получается посетить файл (получить к нему доступ) по разным причинам (например, не получается прочитать атрибуты файла либо не получается открыть папку).

ОК, пока все идет хорошо! Давайте продолжим, взяв на вооружение несколько практических примеров.

Тривиальный обход папки

Имплементация интерфейса `FileVisitor` требует переопределения его четырех методов. Однако API неблокирующего ввода/вывода NIO.2 идет в комплекте со встроенной простой имплементацией этого интерфейса под названием `SimpleFileVisitor`. В простых случаях расширить этот класс будет удобнее, чем имплементировать интерфейс `FileVisitor`, т. к. позволяет переопределять только необходимые методы.

Например, допустим, что мы храним наши онлайн-курсы в подпапках папки `D:/learning`, и мы хотим посетить каждую из этих подпапок посредством API `FileVisitor`. Если во время итеративного обхода по подпапке что-то пойдет не так, мы просто выбросим исключение с сообщением.

Для того чтобы сформировать такое поведение, мы должны переопределить метод `postVisitDirectory()`:

```
class PathVisitor extends SimpleFileVisitor<Path> {
    @Override
    public FileVisitResult postVisitDirectory(
        Path dir, IOException ioe) throws IOException {

        if (ioe != null) {
            throw ioe;
        }

        System.out.println("Visited directory: " + dir);

        return FileVisitResult.CONTINUE;
    }
}
```

Для того чтобы использовать класс `PathVisitor`, нам нужно лишь настроить путь и вызвать один из методов `Files.walkFileTree()` следующим образом (разновидность метода `walkFileTree()`, который применяется здесь, получает начальные файл или папку и соответствующий `FileVisitor`):

```
Path path = Paths.get("D:/learning");
PathVisitor visitor = new PathVisitor();
Files.walkFileTree(path, visitor);
```

Используя приведенный выше фрагмент кода, мы получим такие результаты:

```
Visited directory: D:\learning\books\ajax
Visited directory: D:\learning\books\angular
...
```

Поиск файла по имени

Операция поиска некоторого файла на компьютере встречается очень часто. Как правило, приходится опираться на инструменты, предусмотренные операционной

системой, или применять дополнительные инструменты, но если мы хотим выполнить это программно (например, мы можем написать инструмент поиска файлов со специальными способностями), то `FileVisitor` поможет нам достичь этого довольно простым способом. Вот заготовка такого приложения:

```
public class SearchFileVisitor implements FileVisitor {
    private final Path fileNameToSearch;
    private boolean fileFound;
    ...

    private boolean search(Path file) throws IOException {

        Path fileName = file.getFileName();

        if (fileNameToSearch.equals(fileName)) {
            System.out.println("Искомый файл найден: " +
                fileNameToSearch + " в " + file.toRealPath().toString());

            return true;
        }

        return false;
    }
}
```

Давайте посмотрим на главные контрольные точки и имплементацию поиска файла по имени.

♦ Метод `visitFile()` является нашей главной контрольной точкой. Получив контроль, мы можем запросить у посещенного файла его имя, расширение, атрибуты и т. д. Эта информация необходима для того, чтобы провести сравнение с аналогичной информацией по искомому файлу. Например, мы сравниваем имена и при первом совпадении прекращаем поиск (`TERMINATE`). Но если мы ищем несколько таких файлов (если мы знаем, что их больше одного), мы можем вернуть `CONTINUE`:

```
@Override
public FileVisitResult visitFile(
    Object file, BasicFileAttributes attrs) throws IOException {

    fileFound = search((Path) file);

    if (!fileFound) {
        return FileVisitResult.CONTINUE;
    } else {
        return FileVisitResult.TERMINATE;
    }
}
```



Метод `visitFile()` нельзя использовать для поиска папок. Вместо этого следует вызывать метод `preVisitDirectory()` или метод `postVisitDirectory()`.

- ♦ Метод `visitFileFailed()` является второй важной контрольной точкой. Когда этот метод вызывается, мы знаем, что при посещении текущего файла что-то пошло не так. Мы предпочитаем игнорировать любые подобные проблемы и продолжить (`CONTINUE`) поиск. Бессмысленно останавливать процесс поиска:

```
@Override
public FileVisitResult visitFileFailed(
    Object file, IOException ioe) throws IOException {

    return FileVisitResult.CONTINUE;
}
```

Методы `preVisitDirectory()` и `postVisitDirectory()` не выполняют никаких важных операций, поэтому мы можем пропустить их для краткости.

Для того чтобы начать поиск, мы опираемся на еще одну разновидность метода `Files.walkFileTree()`. На этот раз мы указываем начальную точку поиска (например, все корни), аргументы, которые использовались при поиске (например, переход по символическим ссылкам), максимальное число посещаемых каталожных уровней (например, `Integer.MAX_VALUE`) и `FileVisitor` (например, `SearchFileVisitor`):

```
Path searchFile = Paths.get("JavaModernChallenge.pdf");

SearchFileVisitor searchFileVisitor = new SearchFileVisitor(searchFile);

EnumSet opts = EnumSet.of(FileVisitOption.FOLLOW_LINKS);
Iterable<Path> roots = FileSystems.getDefault().getRootDirectories();

for (Path root: roots) {
    if (!searchFileVisitor.isFileFound()) {
        Files.walkFileTree(root, opts,
            Integer.MAX_VALUE, searchFileVisitor);
    }
}
```

Если вы взглянете на исходный код, который прилагается к этой книге, то приведенный выше поиск будет выполнять рекурсивный обход всех корней (каталогов) вашего компьютера. Приведенный выше пример можно легко адаптировать для поиска по расширению, шаблону или для поиска внутри файлов некоторого текста.

Удаление папки

Прежде чем пытаться удалить папку, мы должны удалить из нее все файлы. Это положение очень важное, т. к. оно не позволяет нам просто вызывать методы `delete()/deleteIfExists()` для папки, содержащей файлы. Изящество решения этой

задачи опирается на имплементацию интерфейса `FileVisitor`, которая начинается со следующей заготовки:

```
public class DeleteFileVisitor implements FileVisitor {
    ...
    private static boolean delete(Path file) throws IOException {
        return Files.deleteIfExists(file);
    }
}
```

Давайте взглянем на главные контрольные точки и имплементацию удаления папки.

- ♦ Метод `visitFile()` является идеальным местом для удаления каждого файла из заданной папки или подпапки (если удалить файл не получается, мы просто передаем его в следующий файл; и не стесняйтесь переделать этот код под собственные нужды):

```
@Override
public FileVisitResult visitFile(
    Object file, BasicFileAttributes attrs) throws IOException {
    delete((Path) file);

    return FileVisitResult.CONTINUE;
}
```

- ♦ Папка может быть удалена только в том случае, если она является пустой, и поэтому метод `postVisitDirectory()` идеален для этого действия (мы игнорируем любое потенциальное исключение `IOException`, но не стесняйтесь переделать этот код под собственные нужды (например, можно занести имена папок, которые не удалось удалить, в журнал, или выбросить исключение, чтобы остановить процесс)):

```
@Override
public FileVisitResult postVisitDirectory(
    Object dir, IOException ioe) throws IOException {
    delete((Path) dir);

    return FileVisitResult.CONTINUE;
}
```

В методах `visitFileFailed()` и `preVisitDirectory()` мы просто возвращаем `CONTINUE`.

Для удаления папки в `D:/learning` мы можем вызвать `DeleteFileVisitor` следующим образом:

```
Path directory = Paths.get("D:/learning");
DeleteFileVisitor deleteFileVisitor = new DeleteFileVisitor();
EnumSet opts = EnumSet.of(FileVisitOption.FOLLOW_LINKS);

Files.walkFileTree(directory, opts, Integer.MAX_VALUE, deleteFileVisitor);
```



Комбинируя `SearchFileVisitor` и `DeleteFileVisitor`, мы можем получить приложение поиска-удаления.

Копирование папки

В копировании файла мы можем опереться на метод `Path copy(Path source, Path target, CopyOption options)` throws `IOException`. Он копирует файл в целевой файл с параметром `options`, указывающим на то, как выполняется копирование.

Комбинируя метод `copy()` со специализированным `FileVisitor`, мы можем скопировать всю папку (включая все ее содержимое). Исходный код заготовки этого специализированного `FileVisitor` приведен ниже:

```
public class CopyFileVisitor implements FileVisitor {
    private final Path copyFrom;
    private final Path copyTo;
    ...

    private static void copySubTree(
        Path copyFrom, Path copyTo) throws IOException {

        Files.copy(copyFrom, copyTo, REPLACE_EXISTING, COPY_ATTRIBUTES);
    }
}
```

Давайте взглянем на главные контрольные точки и имплементацию копирования папки (обратите внимание, что мы будем действовать снисходительно, копируя все, что можем, и избегая исключения, но не стесняйтесь переделать этот код под собственные нужды).

◆ Перед копированием любых файлов из исходной папки нам необходимо скопировать саму исходную папку. Копирование исходной папки (пустой или заполненной) приведет к созданию пустой целевой папки. Эта операция идеально подходит для метода `preVisitDirectory()`:

```
@Override
public FileVisitResult preVisitDirectory(
    Object dir, BasicFileAttributes attrs) throws IOException {

    Path newDir = copyTo.resolve(copyFrom.relativeize((Path) dir));

    try {
        Files.copy((Path) dir, newDir, REPLACE_EXISTING, COPY_ATTRIBUTES);
    } catch (IOException e) {
        System.err.println("Не получается создать "
            + newDir + " [" + e + "]");

        return FileVisitResult.SKIP_SUBTREE;
    }

    return FileVisitResult.CONTINUE;
}
```

- ◆ **Метод `visitFile()` идеален для индивидуального копирования файла:**

```
@Override
public FileVisitResult visitFile(
    Object file, BasicFileAttributes attrs) throws IOException {

    try {
        copySubTree((Path) file, copyTo.resolve(
            copyFrom.relativeTo((Path) file)));
    } catch (IOException e) {
        System.err.println("Не получается скопировать "
            + copyFrom + " [" + e + "]");
    }

    return FileVisitResult.CONTINUE;
}
```

- ◆ **Дополнительно мы можем сохранить атрибуты исходного каталога. Это можно сделать только после того, как файлы были скопированы методом `postVisitDirectory()` (например, давайте сохраним время последнего изменения):**

```
@Override
public FileVisitResult postVisitDirectory(
    Object dir, IOException ioe) throws IOException {

    Path newDir = copyTo.resolve(copyFrom.relativeTo((Path) dir));

    try {
        FileTime time = Files.getLastModifiedTime((Path) dir);
        Files.setLastModifiedTime(newDir, time);
    } catch (IOException e) {
        System.err.println("Не получается сохранить атрибут времени в: "
            + newDir + " [" + e + "]");
    }

    return FileVisitResult.CONTINUE;
}
```

- ◆ **Если посетить файл не получается, то будет вызван метод `visitFileFailed()`. Это хороший момент для того, чтобы обнаружить *циклические ссылки* и сообщить о них. Следуя по ссылкам (`FOLLOW_LINKS`), мы можем столкнуться со случаями, когда дерево файлов имеет *циклическую ссылку* на родительскую папку. Об этих случаях сообщается посредством исключения `FileSystemLoopException` в методе `visitFileFailed()`:**

```
@Override
public FileVisitResult visitFileFailed(
    Object file, IOException ioe) throws IOException {
```

```
if (ioe instanceof FileSystemLoopException) {
    System.err.println("Обнаружен цикл: " + (Path) file);
} else {
    System.err.println("Произошла ошибка, не получается скопировать:"
        + (Path) file + " [" + ioe + "]");
}

return FileVisitResult.CONTINUE;
}
```

Давайте скопируем папку D:/learning/packt в D:/e-courses:

```
Path copyFrom = Paths.get("D:/learning/packt");
Path copyTo = Paths.get("D:/e-courses");

CopyFileVisitor copyFileVisitor = new CopyFileVisitor(copyFrom, copyTo);

EnumSet opts = EnumSet.of(FileVisitOption.FOLLOW_LINKS);

Files.walkFileTree(copyFrom, opts, Integer.MAX_VALUE, copyFileVisitor);
```



Сочетая `CopyFileVisitor` и `DeleteFileVisitor`, мы можем легко сформировать приложение для перемещения папок. В исходном коде, прилагаемом к этой книге, также есть полный пример перемещения папок. Основываясь на опыте, который мы накопили к настоящему моменту, можно с уверенностью сказать, что исходный код должен быть доступным для понимания без дополнительных деталей.

Обратите внимание на журналирование/протоколирование информации о файлах (например, как в случае обработки исключений), т. к. файлы (например, их имена, пути и атрибуты) могут содержать чувствительную информацию, которая может быть использована вредоносным образом.

Метод *Files.walk()* JDK 8

Начиная с JDK 8, класс `Files` был дополнен двумя методами `walk()`. Они возвращают `Stream`, который лениво заполняется объектами `Path`. Он делает это, обходя дерево файлов с корнем в заданном начальном файле, используя заданную максимальную глубину и аргументы:

```
public static Stream<Path> walk(
    Path start, FileVisitOption...options)
    throws IOException

public static Stream<Path> walk(
    Path start, int maxDepth, FileVisitOption...options)
    throws IOException
```

Например, выведем на экран все пути из папки D:/learning, которые начинаются с D:/learning/books/cdi:

```
Path directory = Paths.get("D:/learning");
```

```
Stream<Path> streamOfPath = Files.walk(
    directory, FileVisitOption.FOLLOW_LINKS);

streamOfPath.filter(e -> e.startsWith("D:/learning/books/cdi"))
    .forEach(System.out::println);
```

Теперь вычислим размер папки в байтах (например, папки D:/learning):

```
long folderSize = Files.walk(directory)
    .filter(f -> f.toFile().isFile())
    .mapToLong(f -> f.toFile().length())
    .sum();
```



Этот метод является *слабо согласованным*. Он не замораживает дерево файлов во время итерации. Потенциальные обновления в дереве файлов могут быть не отражены.

135. Наблюдение за путями

Наблюдение за путями на предмет их изменений — это только одна из нитебезопасных целей, которые могут быть достигнуты в JDK 7 посредством низкоуровневого интерфейса `WatchService` в рамках API неблокирующего ввода/вывода NIO.2.

В двух словах, за путем можно наблюдать на предмет его изменений, выполнив два основных шага:

1. Зарегистрировать папку (или папки) для наблюдения за различными типами событий.
2. Когда зарегистрированный тип события обнаруживается службой `WatchService`, оно обрабатывается в отдельной нити исполнения, поэтому служба наблюдения не блокирует.

На уровне API отправной точкой является интерфейс `WatchService`. Он идет в различных вариантах для разных файловых/операционных систем.

Этот интерфейс работает рука об руку с двумя главными классами. Вместе они обеспечивают удобный подход, который можно имплементировать для добавления возможностей наблюдения в некоторый контекст (например, в файловую систему):

- ◆ `Watchable` — любой объект, имплементирующий этот интерфейс, является *наблюдаемым объектом*, и поэтому за ним можно наблюдать на предмет изменений (например, пути);
- ◆ `StandardWatchEventKinds` — этот класс определяет стандартные *типы событий* (это те типы событий, которые мы можем зарегистрировать для уведомлений):
 - `ENTRY_CREATE` — каталожная запись создана;
 - `ENTRY_DELETE` — каталожная запись удалена;

- `ENTRY_MODIFY` — каталожная запись модифицирована; то, что считается модификацией, несколько зависит от платформы, но на самом деле этот тип события всегда должен вызывать модифицирование содержимого файла;
- `OVERFLOW` — специальное событие, указывающее на то, что события, возможно, были потеряны или отброшены.

Служба `WatchService` называется *наблюдателем*, и мы говорим, что *наблюдатель* наблюдает за *наблюдаемыми объектами*. В следующих далее примерах служба `WatchService` будет создана посредством класса `FileSystem`, и она будет следить за зарегистрированным объектом `Path`.

Отслеживание изменений папки

Давайте начнем с заготовки метода, который в качестве аргумента получает `Path` папки, наблюдаемой на предмет изменений:

```
public void watchFolder(Path path)
    throws IOException, InterruptedException {
    ...
}
```

Служба `WatchService` уведомит нас, когда в заданной папке произойдет любое из событий: `ENTRY_CREATE`, `ENTRY_DELETE` или `ENTRY_MODIFY`. Для этого нам нужно выполнить несколько шагов:

Создать службу `WatchService` так, чтобы мы могли контролировать файловую систему. Это осуществляется посредством метода `FileSystem.newWatchService()`:

```
WatchService watchService = FileSystems.getDefault().newWatchService();
```

Зарегистрировать типы событий, которые должны быть уведомлены. Это осуществляется посредством метода `Watchable.register()`:

```
path.register(watchService,
    StandardWatchEventKinds.ENTRY_CREATE,
    StandardWatchEventKinds.ENTRY_MODIFY,
    StandardWatchEventKinds.ENTRY_DELETE);
```



Для каждого *наблюдаемого объекта* мы получаем регистрационный токен в виде экземпляра `WatchKey` (*ключа наблюдения*). Мы получаем этот ключ наблюдения во время регистрации, но служба `WatchService` возвращает соответствующий ключ наблюдения всякий раз, когда событие запускается.

Теперь нам нужно дождаться входящих событий. Это выполняется в бесконечном цикле (когда происходит событие, наблюдатель отвечает за постановку соответствующего *ключа наблюдения* в очередь для последующего извлечения и изменения его статуса на *сигнальный*):

```
while (true) {
    // обработка типов входящих событий
}
```

Теперь нам нужно получить *ключ наблюдения*. Для получения этого ключа предназначены по крайней мере три метода:

- ◆ `poll()` — возвращает следующий ключ из очереди и удаляет его (в противном случае он возвращает `null`, если ключа нет);
- ◆ `poll(long timeout, TimeUnit unit)` — возвращает следующий ключ из очереди и удаляет его; если ключа нет, он ждет в течение указанного таймаута и повторяет попытку. Если ключ по-прежнему недоступен, он возвращает значение `null`;
- ◆ `take()` — возвращает следующий ключ из очереди и удаляет его; если ключа нет, он ждет до тех пор, пока ключ не будет помещен в очередь или бесконечный цикл не будет остановлен:

```
WatchKey key = watchService.take();
```

Далее нам нужно получить ожидающие обработки события *ключа наблюдения*. Ключ наблюдения в *сигнальном состоянии* имеет по крайней мере одно ожидающее обработки событие. Мы можем получить и удалить все события некоторого *ключа наблюдения* посредством метода `WatchKey.pollEvents()` (каждое событие представлено экземпляром класса `WatchEvent`):

```
for (WatchEvent<?> watchEvent : key.pollEvents()) {
    ...
}
```

Затем мы получаем информацию о *типе события*. По каждому событию мы можем получить разную информацию (например, тип события, число возникновений и контекстно-зависимую информацию (скажем, имя файла, вызвавшего событие), которая полезна для обработки события):

```
Kind<?> kind = watchEvent.kind();
WatchEvent<Path> watchEventPath = (WatchEvent<Path>) watchEvent;
Path filename = watchEventPath.context();
```

Далее мы сбрасываем *ключ наблюдения*. Он имеет статус *готовый* (начальный статус при создании), *сигнальный* либо *недействительный*. Получив *сигнальный статус*, ключ наблюдения остается таким до тех пор, пока мы не вызовем метод `reset()`, который попытается вернуть его в состояние *готовности*, чтобы принять состояние события. Если переход из *сигнального статуса* к *готовому статусу* (возобновить ожидание событий) прошел успешно, то метод `reset()` возвращает `true`; в противном случае он возвращает `false`, что означает, что ключ наблюдения может быть *недействительным*. Ключ наблюдения может находиться в *недействительном состоянии*, если он больше не активен (бездействие может быть вызвано явным вызовом метода `close()` *ключа наблюдения*, закрытием наблюдателя, удалением каталога и т. д.):

```
boolean valid = key.reset();

if (!valid) {
    break;
}
```



Когда единственный ключ наблюдения находится в недействительном состоянии, то нет никакой причины оставаться в бесконечном цикле. Следует просто вызвать инструкцию `break`, чтобы прервать цикл.

Наконец, мы закрываем *наблюдателя*. Это делается явным вызовом метода `close()` службы `WatchService` или применением синтаксиса `try` (инструкции `try` с объявлением одного или нескольких ресурсов), как показано ниже:

```
try (WatchService watchService
    = FileSystems.getDefault().newWatchService()) {
    ...
}
```

Исходный код, который прилагается к этой книге, объединяет все эти фрагменты кода в один класс с именем `FolderWatcher`. Результатом будет *наблюдатель*, способный сообщать о событиях создания, удаления и изменения, произошедших на указанном пути.

Для того чтобы наблюдать за путем, а именно `D:/learning/packt`, мы просто вызываем метод `watchFolder()`:

```
Path path = Paths.get("D:/learning/packt");
```

```
FolderWatcher watcher = new FolderWatcher();
watcher.watchFolder(path);
```

При выполнении этого приложения будет показано следующее сообщение:

```
Watching: D:\learning\packt
```

Теперь мы можем создать, удалить или изменить файл непосредственно в этой папке и проверить уведомления. Например, если мы просто скопируем и вставим файл `resources.txt`, то данные на экране будут такими:

```
ENTRY_CREATE -> resources.txt
ENTRY_MODIFY -> resources.txt
```

В конце не забудьте остановить приложение, т. к. оно будет работать бесконечно (теоретически).

Начиная с этого приложения, исходный код, прилагаемый к данной книге, идет еще с двумя приложениями. Одно из них представляет собой симуляцию системы видеозахвата, а другое — симуляцию устройства слежения за лотком принтера. Опираясь на знания, которые вы накопили в этом разделе, оба приложения вам будет довольно легко понять без дальнейших пояснений.

136. Потокковая передача содержимого файла

Задачу потоковой передачи содержимого файла можно решить с помощью JDK 8, используя методы `Files.lines()` и `BufferedReader.lines()`.

Метод `Stream<String> Files.lines(Path path, Charset cs)` читает все строки из файла в качестве потока. Это происходит лениво, по мере того как поток потребляется. Во

время исполнения терминальной операции потока содержимое файла не должно модифицироваться; в противном случае результат не определен.

Рассмотрим пример, который читает содержимое файла `D:/learning/packt/resources.txt` и выводит его на экран (обратите внимание, что мы выполняем код в инструкции `try` с объявлением ресурсов, и поэтому файл закрывается, закрывая поток):

```
private static final String FILE_PATH = "D:/learning/packt/resources.txt";
...
try (Stream<String> filesStream = Files.lines(
    Paths.get(FILE_PATH), StandardCharsets.UTF_8)) {

    filesStream.forEach(System.out::println);
} catch (IOException e) {
    // обработать исключение IOException, если нужно,
    // в противном случае удалить блок catch
}
```

Аналогичный метод без аргументов имеется в классе `BufferedReader`:

```
try (BufferedReader brStream = Files.newBufferedReader(
    Paths.get(FILE_PATH), StandardCharsets.UTF_8)) {

    brStream.lines().forEach(System.out::println);
} catch (IOException e) {
    // обработать исключение IOException, если нужно,
    // в противном случае удалить блок catch
}
```

137. Поиск файлов/папок в дереве файлов

Операция поиска файлов или папок в дереве файлов встречается часто и требуется во многих ситуациях. Благодаря JDK 8 и его новому методу `Files.find()`, мы можем сделать это довольно легко.

Метод `Files.find()` возвращает `Stream<Path>`, который лениво заполняется путями, совпадающими с предоставленными поисковыми ограничениями:

```
public static Stream<Path> find(
    Path start,
    int maxDepth,
    BiPredicate<Path, BasicFileAttributes> matcher,
    FileVisitOption... options
) throws IOException
```

Этот метод действует подобно методу `walk()`, и поэтому он выполняет обход текущего дерева файлов, начиная с заданного пути (`start`) и достигая максимальной заданной глубины (`maxdepth`). Во время итеративного обхода текущего дерева файлов метод применяет заданный предикат (`matcher`). С помощью этого предиката мы за-

даем ограничения, которые должны быть сопоставлены с каждым файлом, входящим в финальный поток. Дополнительно мы можем задать набор аргументов посещения (options).

```
Path startPath = Paths.get("D:/learning");
```

Давайте взглянем на несколько примеров, которые должны прояснить использование этого метода.

- ♦ Найти все файлы, заканчивающиеся расширением `.properties`, и проследовать по *символическим ссылкам*:

```
Stream<Path> resultAsStream = Files.find(
    startPath,
    Integer.MAX_VALUE,
    (path, attr) -> path.toString().endsWith(".properties"),
    FileVisitOption.FOLLOW_LINKS
);
```

- ♦ Найти все обычные файлы, имена которых начинаются с `application`:

```
Stream<Path> resultAsStream = Files.find(
    startPath,
    Integer.MAX_VALUE,
    (path, attr) -> attr.isRegularFile() &&
    path.getFileName().toString().startsWith("application")
);
```

- ♦ Найти все каталоги, созданные после 16 марта 2019 года:

```
Stream<Path> resultAsStream = Files.find(
    startPath,
    Integer.MAX_VALUE,
    (path, attr) -> attr.isDirectory() &&
    attr.creationTime().toInstant()
        .isAfter(LocalDate.of(2019, 3, 16).atStartOfDay()
            .toInstant(ZoneOffset.UTC))
);
```

Если мы предпочитаем выражать ограничения в виде выражения (например, регулярного выражения), то мы можем использовать интерфейс `PathMatcher`. Этот интерфейс идет вместе с методом `matches(Path path)`, который может определить, соответствует ли заданный путь шаблону этого сопоставления.

Имплементация `FileSystem` поддерживает синтаксис *glob* (*) и регулярных выражений (и может поддерживать другой синтаксис) посредством метода `FileSystem.getPathMatcher(String syntaxPattern)`. Ограничения принимают форму *синтаксис:шаблон*.

На основе `PathMatcher` мы можем написать вспомогательные методы, которые способны охватывать широкий диапазон ограничений. Например, следующий ниже

вспомогательный метод извлекает только те файлы, которые удовлетворяют заданному ограничению, указанному как *синтаксис:шаблон*:

```
public static Stream<Path> fetchFilesMatching(Path root,
    String syntaxPattern) throws IOException {

    final PathMatcher matcher
        = root.getFileSystem().getPathMatcher(syntaxPattern);

    return Files.find(root, Integer.MAX_VALUE, (path, attr)
        -> matcher.matches(path) && !attr.isDirectory());
}
```

Отыскать все файлы Java с помощью синтаксиса glob можно следующим образом:

```
Stream<Path> resultAsStream = fetchFilesMatching(startPath, "glob:**/*.java");
```

Если мы хотим лишь перечислить файлы из текущей папки (без каких-либо ограничений и на один уровень глубины), то мы можем опереться на метод `Files.list()`, как показано в следующем ниже примере:

```
try (Stream<Path> allfiles = Files.list(startPath)) {
    ...
}
```

138. Эффективное чтение/запись текстовых файлов

В среде Java эффективное чтение файлов является вопросом выбора правильного подхода. Для лучшего понимания следующего ниже примера давайте допустим, что набором символов по умолчанию нашей платформы является UTF-8. Этот набор символов платформы может быть получен программно посредством метода `Charset.defaultCharset()`.

Сначала мы должны провести различие между сырыми двоичными данными и текстовыми файлами с точки зрения Java. Сырые двоичные данные являются предметом работы двух абстрактных классов, а именно, `InputStream` и `OutputStream`. Для потоковой передачи файлов сырых двоичных данных мы сосредотачиваемся на классах `FileInputStream` и `FileOutputStream`, которые читают/пишут 1 байт (8 бит) за один раз. Для широко известных типов двоичных данных у нас также есть выделенные классы (например, аудиофайл должен обрабатываться посредством `AudioInputStream` вместо `FileInputStream`).

Хотя эти классы делают впечатляющую работу для сырых двоичных данных, они не подходят для текстовых файлов, потому что они являются медленными и могут производить неправильные результаты. Это станет ясным, если мы подумаем, что потоковая передача текстового файла посредством этих классов означает, что из текстового файла считывается и обрабатывается каждый отдельный байт (и такая же утомительная процедура необходима для записи байта). Кроме того, если символ имеет более 1 байта, то можно увидеть странные символы. Другими словами,

декодирование и кодирование 8 бит независимо от набора символов (например, латинского, китайского и т. д.) может привести к неожиданным результатам.

Например, предположим, что у нас есть китайское стихотворение, хранящееся в UTF-16:

```
Path chineseFile = Paths.get("chinese.txt");
```

和毛泽东 << 重上井冈山 >>. 严永欣, 一九八八年.

久有归天愿

终过鬼门关

千里来寻归宿

...

Следующий ниже фрагмент кода не отобразит его, как ожидалось:

```
try (InputStream is = new FileInputStream(chineseFile.toString())) {
    int i;
    while ((i = is.read()) != -1) {
        System.out.print((char) i);
    }
}
```

Таким образом, для того чтобы это исправить, мы должны указать надлежащий набор символов. Хотя `InputStream` это не поддерживает, мы можем опереться на `InputStreamReader` (или соответственно на `OutputStreamReader`). Этот класс является мостом из сырых байтовых потоков к символьным потокам и позволяет нам задавать набор символов:

```
try (InputStreamReader isr = new InputStreamReader(
    new FileInputStream(chineseFile.toFile()),
    StandardCharsets.UTF_16)) {

    int i;
    while ((i = isr.read()) != -1) {
        System.out.print((char) i);
    }
}
```

Все снова в нужном русле, но работает по-прежнему медленно! Теперь приложение может читать более 1 байта за один раз (в зависимости от набора символов) и декодировать их в символы, используя указанный набор символов. Но несколько добавочных байтов по-прежнему не увеличивают скорость.

`InputStreamReader` является мостом между потоками двоичных и символьных данных. Но Java также предусматривает класс `FileReader`. Его цель состоит в том, чтобы устранить этот мост для символьных потоков, которые представлены символьными файлами.

Для текстовых файлов у нас есть выделенный класс `FileReader` (или соответственно `FileWriter`). Он читает 2 или 4 байта (в зависимости от используемого набора символов) за один раз. На самом деле, до JDK 11 `FileReader` не поддерживал явный набор символов. Он просто использовал набор символов по умолчанию платформы. Нам это не подходит, потому что следующий фрагмент кода не даст ожидаемый результат:

```
try (FileReader fr = new FileReader(chineseFile.toFile())) {
    int i;
    while ((i = fr.read()) != -1) {
        System.out.print((char) i);
    }
}
```

Уже в JDK 11 класс `FileReader` дополнен еще двумя конструкторами, которые поддерживают явный набор символов:

- ◆ `FileReader(File file, Charset charset);`
- ◆ `FileReader(String fileName, Charset charset).`

На этот раз мы можем переписать приведенный выше фрагмент кода и получить ожидаемый результат:

```
try (FileReader frch = new FileReader(
    chineseFile.toFile(), StandardCharsets.UTF_16)) {

    int i;
    while ((i = frch.read()) != -1) {
        System.out.print((char) i);
    }
}
```

Чтение 2 или 4 байт за один раз все же лучше, чем чтение 1 байта, но по-прежнему работает медленно. Кроме того, обратите внимание, что приведенные выше решения используют тип `int` для хранения извлеченного символа, и нам нужно явно привести его к типу `char`, чтобы вывести его на экран. В сущности, извлеченный символ из входного файла конвертируется в тип `int`, и мы конвертируем его обратно в тип `char`.

Как раз тут на сцену выходят *буферизующие потоки*. Подумайте о том, что происходит, когда мы смотрим видео онлайн. Пока мы смотрим видео, браузер заранее буферизует входящие байты. Благодаря этому перед нами возникает плавная картинка, т. к. мы видим байты из буфера, что и позволяет избежать потенциальных прерываний, возникающих при просмотре байтов во время передачи по сети (рис. 6.1).

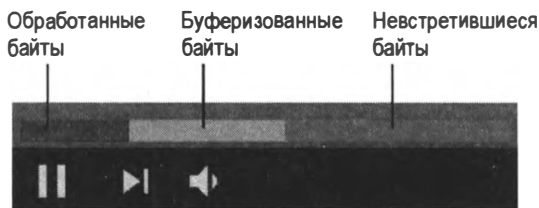


Рис. 6.1

Тот же принцип используется классами, такими как `BufferedInputStream`, `BufferedOutputStream` для сырых двоичных потоков и `BufferedReader` и `BufferedWriter` для символьных потоков. Главная идея заключается в буферизации данных перед обработкой. На этот раз `FileReader` возвращает данные в `BufferedReader` до тех пор, пока они не достигнут конца строки текста (например, `\n` или `\n\r`). `BufferedReader` использует оперативную память для хранения буферизованных данных:

```
try (BufferedReader br = new BufferedReader(
    new FileReader(chineseFile.toFile(), StandardCharsets.UTF_16)) {

    String line;
    // продолжать буферизовать и печатать
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
}
```

Таким образом, вместо чтения 2 байт за один раз мы читаем полную строку текста, что намного быстрее. Этот способ чтения текстовых файлов действительно является эффективным.



Для дальнейшей оптимизации мы можем задать размер буфера с помощью выделенных конструкторов.

Обратите внимание, что класс `BufferedReader` знает, как создавать буфер и работать с ним в контексте входящих данных, но не зависит от источника данных. В нашем примере источником данных является `FileReader`, т. е. файл, но один и тот же класс `BufferedReader` может буферизовать данные из разных источников (например, сети, файла, консоли, принтера, датчика и т. д.). В конце мы читаем то, что буферизовали.

В предыдущих примерах представлены главные подходы к чтению текстовых файлов в среде Java. В JDK 8 был добавлен новый набор методов для облегчения нашей жизни. Для того чтобы создать `BufferedReader`, мы можем также опереться на метод `Files.newBufferedReader(Path path, Charset cs)`:

```
try (BufferedReader br = Files.newBufferedReader(
    chineseFile, StandardCharsets.UTF_16)) {
```

```
String line;
while ((line = br.readLine()) != null) {
    System.out.println(line);
}
}
```

Для класса `BufferedWriter` у нас есть метод `Files.newBufferedWriter()`. Преимущество этих методов заключается в том, что они поддерживают объект `Path` непосредственно.

Для извлечения содержимого текстового файла в качестве `Stream<T>` обратитесь к задаче в разд. 136 *"Потоковая передача содержимого файла"* ранее в этой главе.

Еще одно допустимое решение, которому стоит уделить внимание, приведено ниже:

```
try (BufferedReader br = new BufferedReader(new InputStreamReader(
    new FileInputStream(chineseFile.toFile()), StandardCharsets.UTF_16))) {

    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
}
```

Теперь самое время поговорить о чтении текстовых файлов непосредственно в память.

Чтение текстовых файлов в память

Класс `Files` идет в комплекте с двумя методами, которые могут читать весь текстовый файл в память. Одним из них является `List<String> readAllLines(Path path, Charset cs)`:

```
List<String> lines = Files.readAllLines(
    chineseFile, StandardCharsets.UTF_16);
```

Кроме того, мы можем прочитать все содержимое в строку посредством метода `Files.readString(Path path, Charset cs)`:

```
String content = Files.readString(chineseFile,
    StandardCharsets.UTF_16);
```

Хотя оба метода очень удобны для относительно небольших файлов, они не являются хорошим решением для крупных файлов. Попытка извлечь крупные файлы в память подвержена ошибке `OutOfMemoryError` и, очевидно, будет потреблять много памяти. В качестве альтернативы, в случае с огромными файлами (например, 200 Гб) мы можем сосредоточиться на файлах с отображением в память (`MappedByteBuffer`). Класс `MappedByteBuffer` позволяет нам создавать и изменять огромные файлы и трактовать их как очень большие массивы. Всё выглядит так, будто они находятся в

памяти, даже если это не соответствует действительности. Всё происходит на интуитивном уровне:

```
// либо использовать Files.newByteChannel()
try (FileChannel fileChannel = (FileChannel.open(chineseFile,
    EnumSet.of(StandardOpenOption.READ)))) {

    MappedByteBuffer mbBuffer = fileChannel.map(
        FileChannel.MapMode.READ_ONLY, 0, fileChannel.size());

    if (mbBuffer != null) {
        String bufferContent
            = StandardCharsets.UTF_16.decode(mbBuffer).toString();

        System.out.println(bufferContent);
        mbBuffer.clear();
    }
}
```

Для огромных файлов рекомендуется обходить буфер с фиксированным размером в цикле, как показано ниже:

```
private static final int MAP_SIZE = 5242880; // 5 Мбайт в байтах

try (FileChannel fileChannel = (FileChannel.open(chineseFile,
    EnumSet.of(StandardOpenOption.READ)))) {

    int position = 0;
    long length = fileChannel.size();

    while (position < length) {
        long remaining = length - position;
        int bytestomap = (int) Math.min(MAP_SIZE, remaining);

        MappedByteBuffer mbBuffer = fileChannel.map(
            MapMode.READ_ONLY, position, bytestomap);

        ... // сделать что-то с текущим буфером

        position += bytestomap;
    }
}
```



JDK 13 готовит выпуск неволатильных буферов `MappedByteBuffer`. Так что оставайтесь на линии!

Запись текстовых файлов

Для каждого класса/метода, предназначенного для чтения текстового файла (например, `BufferedReader` и `readString()`), Java предусматривает свой аналог для записи текстового файла (например, `BufferedWriter` и `writeString()`). Вот пример записи текстового файла посредством класса `BufferedWriter`:

```
Path textFile = Paths.get("sample.txt");

try (BufferedWriter bw = Files.newBufferedWriter(
    textFile, StandardCharsets.UTF_8, StandardOpenOption.CREATE,
    StandardOpenOption.WRITE)) {
    bw.write("Lorem ipsum dolor sit amet, ... ");
    bw.newLine();
    bw.write("sed do eiusmod tempor incididunt ...");
}
```

Очень удобный метод записи итерируемого объекта в текстовый файл представлен методом `Files.write(Path path, Iterable<? extends CharSequence> lines, Charset cs, OpenOption... options)`. Например, давайте запишем содержимое списка в текстовый файл (каждый элемент из списка помещается в строку файла):

```
List<String> linesToWrite = Arrays.asList("abc", "def", "ghi");
Path textFile = Paths.get("sample.txt");
Files.write(textFile, linesToWrite, StandardCharsets.UTF_8,
    StandardOpenOption.CREATE, StandardOpenOption.WRITE);
```

Наконец, для того чтобы записать значение типа `String` в файл, мы можем опереться на метод `Files.writeString(Path path, CharSequence csq, OpenOption... options)`:

```
Path textFile = Paths.get("sample.txt");

String lineToWrite = "Lorem ipsum dolor sit amet, ...";
Files.writeString(textFile, lineToWrite, StandardCharsets.UTF_8,
    StandardOpenOption.CREATE, StandardOpenOption.WRITE);
```



TIP

Посредством перечисления `StandardOpenOption` (как подкласса класса перечислений `Enum`) мы можем контролировать то, как файл открывается. В предыдущих примерах файлы создавались, если они не существовали (`CREATE`), и открывались с доступом для записи (`WRITE`). Существует много других вариантов (например, `APPEND`, `DELETE_ON_CLOSE` и т. д.).

Наконец, запись текстового файла посредством класса `MappedByteBuffer` может быть выполнена следующим образом (это может быть полезно для записи огромных текстовых файлов):

```
Path textFile = Paths.get("sample.txt");
CharBuffer cb = CharBuffer.wrap("Lorem ipsum dolor sit amet, ...");

try (FileChannel fileChannel = (FileChannel) Files.newByteChannel(
    textFile, EnumSet.of(StandardOpenOption.CREATE,
        StandardOpenOption.READ, StandardOpenOption.WRITE))) {
```

```
MappedByteBuffer mbBuffer = fileChannel
    .map(FileChannel.MapMode.READ_WRITE, 0, cb.length());

if (mbBuffer != null) {
    mbBuffer.put(StandardCharsets.UTF_8.encode(cb));
}
}
```

139. Эффективное чтение/запись двоичных файлов

В предыдущей задаче из *разд. "Эффективное чтение/запись текстовых файлов"* мы говорили о *буферизации потоковой передачи* (для ясности картины рекомендуется прочитать указанную задачу перед этой). То же самое происходит и с двоичными файлами, поэтому мы можем перейти непосредственно к нескольким примерам.

Рассмотрим следующий ниже двоичный файл и его размер в байтах:

```
Path binaryFile = Paths.get("build/classes/modern/challenge/Main.class");

int fileSize = (int) Files.readAttributes(
    binaryFile, BasicFileAttributes.class).size();
```

Мы можем прочитать содержимое файла в массив `byte[]` посредством класса `FileInputStream` (в нем буферизация не используется):

```
final byte[] buffer = new byte[fileSize];

try (InputStream is = new FileInputStream(binaryFile.toString())) {
    int i;
    while ((i = is.read(buffer)) != -1) {
        System.out.print("\nReading ... ");
    }
}
```

Однако приведенный пример не очень эффективен. Достигнуть высокую эффективность при чтении `buffer.length` байт из этого входного потока в байтовый массив можно посредством класса `BufferedInputStream` следующим образом:

```
final byte[] buffer = new byte[fileSize];

try (BufferedInputStream bis = new BufferedInputStream(
    new FileInputStream(binaryFile.toFile()))) {

    int i;
    while ((i = bis.read(buffer)) != -1) {
        System.out.print("\nReading ... " + i);
    }
}
```

Класс `FileInputStream` можно также получить с помощью метода `Files.newInputStream()`. Преимущество этого метода состоит в том, что он поддерживает объект `Path` непосредственно:

```
final byte[] buffer = new byte[fileSize];

try (BufferedInputStream bis = new BufferedInputStream(
    Files.newInputStream(binaryFile))) {

    int i;
    while ((i = bis.read(buffer)) != -1) {
        System.out.print("\nReading ... " + i);
    }
}
```

Если файл является слишком крупным, чтобы поместиться в буфер размера файла, то предпочтительно читать его через меньший буфер с фиксированным размером (например, 512 байт) и с помощью разновидности метода `read()`, а именно:

- ◆ `read(byte[] b);`
- ◆ `read(byte[] b, int off, int len);`
- ◆ `readNBytes(byte[] b, int off, int len);`
- ◆ `readNBytes(int len);`



Метод `read()` без аргументов будет читать входной поток побайтно. Этот способ является самым неэффективным, в особенности без использования буферизации.

В качестве альтернативы, если наша цель — прочитать входной поток как байтовый массив, то мы можем опереться на класс `ByteArrayInputStream` (он использует внутренний буфер, поэтому нет необходимости применять класс `BufferedInputStream`):

```
final byte[] buffer = new byte[fileSize];

try (ByteArrayInputStream bais = new ByteArrayInputStream(buffer)) {

    int i;
    while ((i = bais.read(buffer)) != -1) {
        System.out.print("\nЧитается ... ");
    }
}
```

Предыдущие подходы хороши для сырых двоичных данных, но иногда наши двоичные файлы содержат особенные данные (например, значения типа `int`, `float` и т. д.). В таких случаях классы `DataInputStream` и `DataOutputStream` предусматривают удобные методы для чтения и записи некоторых типов данных. Будем считать, что

у нас есть файл `data.bin`, который содержит числа с плавающей точкой. Мы можем эффективно прочитать его следующим образом:

```
Path dataFile = Paths.get("data.bin");

try (DataInputStream dis = new DataInputStream(
    new BufferedInputStream(Files.newInputStream(dataFile)))) {

    while (dis.available() > 0) {
        float nr = dis.readFloat();
        System.out.println("Read: " + nr);
    }
}
```



Оба класса являются всего лишь двумя фильтрами данных, предусмотренными средой Java. Для обзора всех поддерживаемых фильтров данных ознакомьтесь с подклассами класса `FilterInputStream`. Кроме того, класс `Scanner` является хорошей альтернативой для чтения некоторых типов данных. Дополнительную информацию см. в разд. 148 *"Работа с классом Scanner"* далее в этой главе.

Теперь посмотрим, как читать двоичные файлы непосредственно в память.

Чтение двоичных файлов в память

Прочитать весь двоичный файл в память можно посредством метода `Files.readAllBytes()`:

```
byte[] bytes = Files.readAllBytes(binaryFile);
```

Аналогичный метод существует и в классе `InputStream`.

Хотя эти методы очень удобны для относительно небольших файлов, они не являются хорошим вариантом решения для крупных файлов. Попытка извлечь крупные файлы в память подвержена ошибке `OutOfMemoryError` и, очевидно, будет потреблять много памяти. В качестве альтернативы, в случае с огромными файлами (например, 200 Гб) мы можем сосредоточиться на файлах с отображением в память (`MappedByteBuffer`). Класс `MappedByteBuffer` позволяет нам создавать и изменять огромные файлы и трактовать их как очень большой массив. Всё выглядит так, как будто они находятся в памяти, даже если это не соответствует действительности. Всё происходит на интуитивном уровне:

```
try (FileChannel fileChannel = (FileChannel.open(binaryFile,
    EnumSet.of(StandardOpenOption.READ)))) {

    MappedByteBuffer mbBuffer = fileChannel.map(
        FileChannel.MapMode.READ_ONLY, 0, fileChannel.size());

    System.out.println("\nЧтение: " + mbBuffer.limit() + " байт");
}
```

Для огромных файлов рекомендуется обходить буфер с фиксированным размером в цикле следующим образом:

```
private static final int MAP_SIZE = 5242880;    // 5 Мбайт в байты

try (FileChannel fileChannel = FileChannel.open(
    binaryFile, StandardOpenOption.READ)) {

    int position = 0;
    long length = fileChannel.size();

    while (position < length) {
        long remaining = length - position;
        int bytestomap = (int) Math.min(MAP_SIZE, remaining);

        MappedByteBuffer mbBuffer = fileChannel.map(
            MapMode.READ_ONLY, position, bytestomap);

        ... // сделать что-то с текущим буфером

        position += bytestomap;
    }
}
```

Запись двоичных файлов

Эффективным способом записи двоичных файлов является использование класса `BufferedOutputStream`. Например, записать массив `byte[]` в файл можно следующим образом:

```
final byte[] buffer...;

Path classFile = Paths.get("build/classes/modern/challenge/Main.class");

try (BufferedOutputStream bos = new BufferedOutputStream(
    Files.newOutputStream(classFile, StandardOpenOption.CREATE,
        StandardOpenOption.WRITE))) {
    bos.write(buffer);
}
```



Если вы пишете побайтно, то следует использовать метод `write(int b)`, а если вы записываете фрагмент данных, то метод `write(byte[] b, int off, int len)`.

Очень удобный способ записи массива `byte[]` в файл представлен методом `Files.write(Path path, byte[] bytes, OpenOption... options)`.

Например, давайте запишем содержимое предыдущего буфера:

```
Path classFile = Paths.get("build/classes/modern/challenge/Main.class");
```

```
Files.write(classFile, buffer,  
    StandardOpenOption.CREATE, StandardOpenOption.WRITE);
```

Записать двоичный файл посредством класса `MappedByteBuffer` можно следующим образом (это лайфхак для записи огромных текстовых файлов):

```
Path classFile = Paths.get("build/classes/modern/challenge/Main.class");
```

```
try (FileChannel fileChannel = (FileChannel) Files.newByteChannel(  
    classFile, EnumSet.of(StandardOpenOption.CREATE,  
        StandardOpenOption.READ, StandardOpenOption.WRITE))) {
```

```
    MappedByteBuffer mbBuffer = fileChannel  
        .map(FileChannel.MapMode.READ_WRITE, 0, buffer.length);
```

```
    if (mbBuffer != null) {  
        mbBuffer.put(buffer);  
    }
```

```
}
```

Наконец, если мы пишем определенный фрагмент данных (не сырые двоичные данные), то можем опереться на класс `DataOutputStream`. Он идет в комплекте с методами `writeFoo()` для разных типов данных. Например, запишем несколько чисел с плавающей точкой в файл:

```
Path floatFile = Paths.get("float.bin");
```

```
try (DataOutputStream dis = new DataOutputStream(  
    new BufferedOutputStream(Files.newOutputStream(floatFile)))) {  
    dis.writeFloat(23.56f);  
    dis.writeFloat(2.516f);  
    dis.writeFloat(56.123f);  
}
```

140. Поиск в больших файлах

Операция поиска и подсчета числа появлений того или иного строкового значения в файле встречается очень часто. Выполнение этой операции как можно быстрее — обязательное требование, в особенности если файл является большим (например, 200 Гб).

Обратите внимание, что следующие далее имплементации исходят из того, что строковое значение `ll` встречается в `lll` только один раз, а не два раза. Кроме того,

первые три имплементации опираются на следующий ниже вспомогательный метод из *разд. 17 "Подсчет числа появлений подстроки в строке" главы 1*:

```
private static int countStringInString(String string, String tofind) {  
    return string.split(Pattern.quote(tofind), -1).length - 1;  
}
```

С учетом сказанного, давайте взглянем на несколько подходов к этой задаче.

Решение на основе класса *BufferedReader*

Из предыдущих задач мы уже знаем, что *BufferedReader* является очень эффективным для чтения текстовых файлов. Поэтому мы можем использовать его и для чтения большого файла. Во время чтения для каждой строки, получаемой посредством метода *BufferedReader.readLine()*, нам нужно подсчитать число появлений искомой строки посредством метода *countStringInString()*:

```
public static int countOccurrences(Path path, String text, Charset ch)  
    throws IOException {  
  
    int count = 0;  
  
    try (BufferedReader br = Files.newBufferedReader(path, ch)) {  
        String line;  
        while ((line = br.readLine()) != null) {  
            count += countStringInString(line, text);  
        }  
    }  
  
    return count;  
}
```

Решение на основе метода *Files.readAllLines()*

Если оперативная память (RAM) не является нашей проблемой, то мы можем попытаться прочитать весь файл в память (посредством метода *Files.readAllLines()*) и обработать его оттуда. Наличие всего файла в памяти поддерживает параллельную обработку. Поэтому, если наше оборудование допускает параллельную обработку, то мы можем попытаться применить метод *parallelStream()*, как показано ниже:

```
public static int countOccurrences(Path path, String text, Charset ch)  
    throws IOException {  
  
    return Files.readAllLines(path, ch).parallelStream()  
        .mapToInt((p) -> countStringInString(p, text))  
        .sum();  
}
```



Если метод `parallelStream()` не приносит никаких преимуществ, то мы можем переключиться на `stream()`. Это просто вопрос сравнительного анализа методов.

Решение на основе метода *Files.lines()*

Мы можем попытаться воспользоваться потоками посредством метода `Files.lines()`. На этот раз мы получаем файл в виде ленивого потока `Stream<String>`. Если мы можем воспользоваться преимуществами параллельной обработки (сравнительный анализ показывает более высокие характеристики), то очень просто параллелизовать `Stream<String>`, вызвав метод `parallel()`:

```
public static int countOccurrences(Path path, String text, Charset ch)
    throws IOException {

    return Files.lines(path, ch).parallel()
        .mapToInt((p) -> countStringInString(p, text))
        .sum();
}
```

Решение на основе класса *Scanner*

Начиная с JDK 9, класс `Scanner` идет в комплекте с методом, который возвращает поток отделенных разделителями токенов, `Stream<String> tokens()`. Если мы рассматриваем поисковый текст как разделитель класса `Scanner` и подсчитываем элементы потока, возвращаемые методом `tokens()`, то мы получаем правильный результат:

```
public static long countOccurrences(
    Path path, String text, Charset ch) throws IOException {

    long count;

    try (Scanner scanner = new Scanner(path, ch)
        .useDelimiter(Pattern.quote(text))) {

        count = scanner.tokens().count() - 1;
    }

    return count;
}
```



Конструкторы для класса `Scanner`, которые поддерживают явно задаваемый набор символов, были добавлены в JDK 10.

Решение на основе класса *MappedByteBuffer*

Последний вариант решения, который мы здесь упомянем, основан на классах `MappedByteBuffer` и `FileChannel` API неблокирующего ввода/вывода NIO.2. Это программное решение открывает отображенный в память байтовый буфер (`MappedByteBuffer`) из канала `FileChannel` для заданного файла. Мы обходим выбранный байтовый буфер в цикле и ищем совпадения с искомым строковым значением (оно конвертируется в массив `byte[]`, и поиск происходит побайтно).

Для малых файлов это решение работает быстрее при загрузке всего файла в память. Для крупных/огромных файлов быстрее загружать и обрабатывать файлы фрагментами (например, размером 5 Мб). После загрузки фрагмента мы должны подсчитать число появлений искомого строкового значения. Мы сохраняем результат и передаем его следующему фрагменту данных. И повторяем это до тех пор, пока не обработаем весь файл.

Давайте взглянем на ключевую часть этой имплементации (для получения полного исходного текста программы обратитесь к исходному коду, прилагаемому к этой книге):

```
private static final int MAP_SIZE = 5242880; // 5 Мб в байтах

public static int countOccurrences(Path path, String text)
    throws IOException {

    final byte[] texttofind = text.getBytes(StandardCharsets.UTF_8);
    int count = 0;

    try (FileChannel fileChannel = FileChannel.open(path,
        StandardOpenOption.READ)) {

        int position = 0;
        long length = fileChannel.size();

        while (position < length) {
            long remaining = length - position;
            int bytestomap = (int) Math.min(MAP_SIZE, remaining);

            MappedByteBuffer mbBuffer = fileChannel.map(
                MapMode.READ_ONLY, position, bytestomap);

            int limit = mbBuffer.limit();
            int lastSpace = -1;
            int firstChar = -1;
```

```

while (mbBuffer.hasRemaining()) {
    // спагетти-код опущен для краткости
    ...
}
}

return count;
}

```

Это решение чрезвычайно быстрое, поскольку файл читается непосредственно из памяти операционной системы без необходимости загрузки в JVM. Операции выполняются на нативном уровне, именуемом уровнем операционной системы. Обратите внимание, что эта имплементация работает только для набора символов UTF-8, но она может быть переделана и для других наборов символов.

141. Чтение файла JSON/CSV как объекта

Файлы JSON и CSV теперь можно встретить повсюду. Операция чтения (десериализация) файлов JSON/CSV может выполняться на повседневной основе, и она обычно предшествует нашей бизнес-логике. Операция записи (сериализации) файлов JSON/CSV также является популярной, и она обычно выполняется в конце бизнес-логики. Между чтением и записью таких файлов приложение использует данные в качестве объектов.

Чтение/запись файла JSON как объекта

Начнем с трех текстовых файлов, представляющих собой типичные JSON-подобные попарные связки (рис. 6.2).

Сырой JSON	Массив-подобный JSON	Отображение-подобный JSON
<pre>{ "type": "Gac", "weight": 2000}, "type": "Hemi", "weight": 1200} </pre>	<pre>[{ "type": "Gac", "weight": 2000 }, { "type": "Hemi", "weight": 1200 }] </pre>	<pre>{ "A": { "type": "Gac", "weight": 2000 }, "B": { "type": "Hemi", "weight": 1200 } } </pre>
melons_raw.json	melons_array.json	melons_map.json

Рис. 6.2

В `melons_raw.json` у нас один элемент JSON на строку. Каждая строка является фрагментом JSON, который не зависит от предыдущей строки, но имеет ту же схему. В `melons_array.json` у нас массив JSON, и в `melons_map.json` у нас JSON, который хорошо вписывается в отображение Map, используемое в среде Java.

Для каждого из этих файлов у нас есть объект Path, как показано ниже:

```
Path pathArray = Paths.get("melons_array.json");
Path pathMap = Paths.get("melons_map.json");
Path pathRaw = Paths.get("melons_raw.json");
```

Теперь давайте рассмотрим три выделенные библиотеки для чтения содержимого этих файлов в виде экземпляров класса Melon:

```
public class Melon {

    private String type;
    private int weight;

    // геттеры и сеттеры опущены для краткости
}
```

Использование библиотеки JSON-B

Java EE 8 идет в комплекте с JAXB-подобной декларативной привязкой JSON, именуемой JSON-B (JSR-367). Привязка JSON-B совместима с JAXB и другими API Java EE/SE. Jakarta EE поднимает Java EE 8 JSON (P и B) на следующий уровень. Его API выставляется наружу посредством классов javax.json.bind.Jsonb и javax.json.bind.JsonbBuilder:

```
Jsonb jsonb = JsonbBuilder.create();
```

Для десериализации мы используем метод Jsonb.fromJson(), в то время как для сериализации применяем метод Jsonb.toJson().

◆ Прочитать melons_array.json как массив Array объектов класса Melon:

```
Melon[] melonsArray = jsonb.fromJson(Files.newBufferedReader(
    pathArray, StandardCharsets.UTF_8), Melon[].class);
```

◆ Прочитать melons_array.json как список List объектов класса Melon:

```
List<Melon> melonsList = jsonb.fromJson(Files.newBufferedReader(
    pathArray, StandardCharsets.UTF_8), ArrayList.class);
```

◆ Прочитать melons_map.json как отображение Map объектов класса Melon:

```
Map<String, Melon> melonsMap = jsonb.fromJson(Files.newBufferedReader(
    pathMap, StandardCharsets.UTF_8), HashMap.class);
```

◆ Прочитать melons_raw.json построчно в отображение Map:

```
Map<String, String> stringMap = new HashMap<>();

try (BufferedReader br = Files.newBufferedReader(
    pathRaw, StandardCharsets.UTF_8)) {

    String line;

    while ((line = br.readLine()) != null) {
        stringMap = jsonb.fromJson(line, HashMap.class);
    }
}
```

```
System.out.println("Текущее отображение: " + stringMap);
}
}
```

◆ Прочитать `melons_raw.json` построчно в объект `Melon`:

```
try (BufferedReader br = Files.newBufferedReader(
    pathRaw, StandardCharsets.UTF_8)) {

    String line;

    while ((line = br.readLine()) != null) {
        Melon melon = jsonb.fromJson(line, Melon.class);
        System.out.println("Текущая дыня: " + melon);
    }
}
```

◆ Записать объект в JSON-файл (`melons_output.json`):

```
Path path = Paths.get("melons_output.json");

jsonb.toJson(melonsMap, Files.newBufferedWriter(path,
    StandardCharsets.UTF_8, StandardOpenOption.CREATE,
    StandardOpenOption.WRITE));
```

Использование библиотеки Jackson

Jackson — это популярная и быстрая библиотека, предназначенная для обработки (сериализации/десериализации) данных JSON. API библиотеки Jackson опирается на `com.fasterxml.jackson.databind.ObjectMapper`. Давайте снова пройдемся по приведенным выше примерам, но на этот раз с использованием библиотеки Jackson:

```
ObjectMapper mapper = new ObjectMapper();
```

Для десериализации мы используем метод `ObjectMapper.readValue()`, в то время как для сериализации — метод `ObjectMapper.writeValue()`.

◆ Прочитать `melons_array.json` как массив `Array` объектов класса `Melon`:

```
Melon[] melonsArray = mapper.readValue(Files.newBufferedReader(
    pathArray, StandardCharsets.UTF_8), Melon[].class);
```

◆ Прочитать `melons_array.json` как список `List` объектов класса `Melon`:

```
List<Melon> melonsList = mapper.readValue(Files.newBufferedReader(
    pathArray, StandardCharsets.UTF_8), ArrayList.class);
```

◆ Прочитать `melons_map.json` как отображение `Map` объектов класса `Melon`:

```
Map<String, Melon> melonsMap = mapper.readValue(Files.newBufferedReader(
    pathMap, StandardCharsets.UTF_8), HashMap.class);
```

◆ Прочитать `melons_raw.json` построчно в отображение `Map`:

```
Map<String, String> stringMap = new HashMap<>();

try (BufferedReader br = Files.newBufferedReader(
    pathRaw, StandardCharsets.UTF_8)) {
```

```
String line;

while ((line = br.readLine()) != null) {
    stringMap = mapper.readValue(line, HashMap.class);
    System.out.println("Текущее отображение: " + stringMap);
}
}
```

◆ Прочитать melons_raw.json построчно в объект Melon:

```
try (BufferedReader br = Files.newBufferedReader(
    pathRaw, StandardCharsets.UTF_8)) {

    String line;

    while ((line = br.readLine()) != null) {
        Melon melon = mapper.readValue(line, Melon.class);
        System.out.println("Текущая дыня: " + melon);
    }
}
```

◆ Записать объект в JSON-файл (melons_output.json):

```
Path path = Paths.get("melons_output.json");

mapper.writeValue(Files.newBufferedWriter(path,
    StandardCharsets.UTF_8, StandardOpenOption.CREATE,
    StandardOpenOption.WRITE), melonsMap);
```

Использование библиотеки Gson

Gson — это еще одна быстрая библиотека, предназначенная для обработки (сериализации/десериализации) данных JSON. В проект Maven она может быть добавлена как зависимость в pom.xml. Ее API опирается на имя класса com.google.gson.Gson. Исходный код, прилагаемый к этой книге, предоставляет ряд примеров с ее использованием.

Чтение CSV-файла как объекта

Самый простой CSV-файл выглядит так, как показано на рис. 6.3 (элементы данных разделены запятыми).

melon.csv
CSV
Gaas,2000
Hemi,1500
Cantaloupe,800
Golden Prize,2300
Crenshaw,3000

Рис. 6.3

Простое и эффективное решение для десериализации такого рода CSV-файла основывается на классе `BufferedReader` и методе `String.split()`. Мы можем прочитать каждую строку файла посредством метода `BufferedReader.readLine()` и разделить ее с помощью разделителя-запятой с помощью метода `String.split()`. Результат (каждая строка содержимого) может храниться в `List<String>`. Финальным результатом будет `List<List<String>>`, как показано ниже:

```
public static List<List<String>> readAsObject(
    Path path, Charset cs, String delimiter) throws IOException {

    List<List<String>> content = new ArrayList<>();

    try (BufferedReader br = Files.newBufferedReader(path, cs)) {
        String line;

        while ((line = br.readLine()) != null) {
            String[] values = line.split(delimiter);
            content.add(Arrays.asList(values));
        }

    }

    return content;
}
```

Если данные CSV имеют соответствия в обыкновенном объекте Java (POJO) (например, наш CSV является результатом сериализации группы экземпляров класса `Melon`), то они могут быть десериализованы, как показано в следующем ниже примере:

```
public static List<Melon> readAsMelon(
    Path path, Charset cs, String delimiter) throws IOException {

    List<Melon> content = new ArrayList<>();

    try (BufferedReader br = Files.newBufferedReader(path, cs)) {
        String line;

        while ((line = br.readLine()) != null) {
            String[] values = line.split(Pattern.quote(delimiter));
            content.add(new Melon(values[0], Integer.valueOf(values[1])));
        }

    }

    return content;
}
```



Для сложных CSV-файлов рекомендуется использовать специальные библиотеки (например, `OpenCSV`, `Apache Commons CSV`, `Super CSV` и т. д.).

142. Работа с временными файлами/папками

API неблокирующего ввода/вывода NIO.2 предусматривает поддержку работы с временными папками/файлами. Например, мы можем легко отыскать месторасположение по умолчанию временных папок/файлов следующим образом:

```
String defaultBaseDir = System.getProperty("java.io.tmpdir");
```

Обычно в Windows заданными по умолчанию временными папками являются папки C:\Temp, %Windows%\Temp, или временный каталог для каждого пользователя в Local Settings\Temp (это месторасположение обычно управляется с помощью переменной среды TEMP). В Linux/UNIX глобальными временными каталогами являются /tmp и /var/tmp. Приведенная выше строка кода будет возвращать месторасположение по умолчанию в зависимости от операционной системы.

В следующем далее разделе мы узнаем, как создавать временную папку или временный файл.

Создание временных папки или файла

Создать *временную папку* можно посредством метода `Path createTempDirectory(Path dir, String prefix, FileAttribute<?>... attrs)`. Это статический метод в классе `Files`, который можно использовать следующим образом.

- ◆ Создать временную папку без префикса месторасположения по умолчанию операционной системы (ОС):

```
// C:\Users\Anghel\AppData\Local\Temp\8083202661590940905
Path tmpNoPrefix = Files.createTempDirectory(null);
```

- ◆ Создать временную папку со специализированным префиксом в заданном по умолчанию месторасположении ОС:

```
// C:\Users\Anghel\AppData\Local\Temp\logs_5825861687219258744
String customDirPrefix = "logs_";
Path tmpCustomPrefix = Files.createTempDirectory(customDirPrefix);
```

- ◆ Создать временную папку со специализированным префиксом в специализированном месторасположении:

```
// D:\tmp\logs_10153083118282372419
Path customBaseDir = FileSystems.getDefault().getPath("D:/tmp");
String customDirPrefix = "logs_";
Path tmpCustomLocationAndPrefix
    = Files.createTempDirectory(customBaseDir, customDirPrefix);
```

Создать *временный файл* можно посредством метода `Path createTempFile(Path dir, String prefix, String suffix, FileAttribute<?>... attrs)`. Это статический метод в классе `Files`, который можно использовать следующим образом.

- ◆ Создать временный файл без префикса и суффикса в заданном по умолчанию месторасположении ОС:

```
// C:\Users\Anghel\AppData\Local\Temp\16106384687161465188.tmp
Path tmpNoPrefixSuffix = Files.createTempFile(null, null);
```

- ◆ Создать временный файл со специализированным префиксом и суффиксом в заданном по умолчанию месторасположении ОС:

```
// C:\Users\Anghel\AppData\Local\Temp\log_402507375350226.txt
String customFilePrefix = "log_";
String customFileSuffix = ".txt";
Path tmpCustomPrefixAndSuffix
    = Files.createTempFile(customFilePrefix, customFileSuffix);
```

- ◆ Создать временный файл со специализированным префиксом и суффиксом в специализированном месторасположении:

```
// D:\tmp\log_13299365648984256372.txt
Path customBaseDir = FileSystems.getDefault().getPath("D:/tmp");
String customFilePrefix = "log_";
String customFileSuffix = ".txt";
Path tmpCustomLocationPrefixSuffix = Files.createTempFile(
    customBaseDir, customFilePrefix, customFileSuffix);
```

В следующих далее разделах мы рассмотрим разные способы удаления временной папки или временного файла.

Удаление временной папки или временного файла посредством перехватчика отключения

Удаление временной папки и/или временного файла может быть выполнено с помощью операционной системы либо специализированных инструментов. Однако иногда нам нужно управлять этой операцией программно и удалять папку/файл, основываясь на разных конструктивных соображениях.

Решение этой задачи зависит от механизма *перехватчика отключения* (shutdown-hook), который может быть имплементирован посредством метода `Runtime.getRuntime().addShutdownHook()`. Этот механизм полезен всякий раз, когда нам нужно выполнить те или иные операции (например, операции очистки) непосредственно перед завершением работы JVM. Он имплементирован как нить исполнения Java, чей метод `run()` вызывается, когда перехватчик отключения выполняется JVM-машиной при завершении работы. Это показано в следующем ниже фрагменте кода:

```
Path customBaseDir = FileSystems.getDefault().getPath("D:/tmp");
String customDirPrefix = "logs_";
String customFilePrefix = "log_";
String customFileSuffix = ".txt";

try {
    Path tmpDir = Files.createTempDirectory(
        customBaseDir, customDirPrefix);
    Path tmpFile1 = Files.createTempFile(
        tmpDir, customFilePrefix, customFileSuffix);
```



```

Path tmpFile2 = Files.createTempFile(
    tmpDir, customFilePrefix, customFileSuffix);

Runtime.getRuntime().addShutdownHook(new Thread() {
    @Override
    public void run() {
        try (DirectoryStream<Path> ds = Files.newDirectoryStream(tmpDir)) {
            for (Path file: ds) {
                Files.delete(file);
            }

            Files.delete(tmpDir);
        } catch (IOException e) {
            ...
        }
    }
});

// просимулировать некие операции с временным файлом, пока он не будет удален
Thread.sleep(10000);
} catch (IOException | InterruptedException e) {
    ...
}

```



Перехватчик отключения не будет исполняться в случае ненормальных/принудительных завершений (например, при сбоях JVM, срабатывании терминальных операций и т. д.). Он запускается, когда все нити исполнения заканчиваются или когда вызывается `System.exit(0)`. Рекомендуется обеспечивать его быструю работу, т. к. он может быть остановлен принудительно до завершения, если что-то пойдет не так (например, ОС выключается). Программно *перехватчик отключения* может быть остановлен только методом `Runtime.halt()`.

Удаление временной папки или временного файла посредством метода `deleteOnExit()`

Еще один вариант решения задачи удаления временной папки или временного файла опирается на метод `File.deleteOnExit()`. Вызывая этот метод, мы можем зарегистрироваться для удаления папки/файла. Действие удаления происходит, когда JVM отключается:

```

Path customBaseDir = FileSystems.getDefault().getPath("D:/tmp");
String customDirPrefix = "logs_";
String customFilePrefix = "log_";
String customFileSuffix = ".txt";

try {
    Path tmpDir = Files.createTempDirectory(
        customBaseDir, customDirPrefix);

```

```
System.out.println("Created temp folder as: " + tmpDir);
Path tmpFile1 = Files.createTempFile(
    tmpDir, customFilePrefix, customFileSuffix);
Path tmpFile2 = Files.createTempFile(
    tmpDir, customFilePrefix, customFileSuffix);

try (DirectoryStream<Path> ds = Files.newDirectoryStream(tmpDir)) {
    tmpDir.toFile().deleteOnExit();

    for (Path file: ds) {
        file.toFile().deleteOnExit();
    }
} catch (IOException e) {
    ...
}

// просимулировать некие операции с временным файлом, пока он не будет удален
Thread.sleep(10000);
} catch (IOException | InterruptedException e) {
    ...
}
```



Рекомендуется опираться на метод `deleteOnExit()`, только когда приложение управляет небольшим числом временных папок/файлов. Этот метод потребляет много памяти (ему требуется память для каждого временного ресурса, зарегистрированного для удаления), и эта память не может быть высвобождена до отключения JVM. Обратите внимание на то, что этот метод должен быть вызван для регистрации каждого временного ресурса, а удаление происходит в обратном порядке регистрации (например, мы должны зарегистрировать временную папку до регистрации ее содержимого).

Удаление временного файла посредством аргумента **DELETE_ON_CLOSE**

Еще одно решение в том, что касается удаления временного файла, опирается на `StandardOpenOption.DELETE_ON_CLOSE` (удаляет файл при закрытии потока). Например, следующий ниже фрагмент кода создает временный файл посредством метода `createTempFile()` и открывает для него буферизованный пишущий поток с явно заданным аргументом `DELETE_ON_CLOSE`:

```
Path customBaseDir = FileSystems.getDefault().getPath("D:/tmp");
String customFilePrefix = "log_";
String customFileSuffix = ".txt";
Path tmpFile = null;

try {
    tmpFile = Files.createTempFile(
        customBaseDir, customFilePrefix, customFileSuffix);
```

```
} catch (IOException e) {  
    ...  
}  
  
try (BufferedWriter bw = Files.newBufferedWriter(tmpFile,  
    StandardCharsets.UTF_8, StandardOpenOption.DELETE_ON_CLOSE)) {  
  
    // просимулировать некие операции с временным файлом, пока он не будет удален  
    Thread.sleep(10000);  
} catch (IOException | InterruptedException e) {  
    ...  
}
```



Это решение может быть принято на вооружение для любого файла. Оно не относится к временным ресурсам.

143. Фильтрация файлов

Операция фильтрации файлов из объекта `Path` встречается очень часто. Например, нам могут потребоваться файлы только конкретного типа, с некоторым шаблоном имени, модифицированные сегодня и т. д.

Фильтрация посредством метода *Files.newDirectoryStream()*

Без какого-либо фильтра мы можем легко обойти содержимое папки в цикле (на один уровень в глубину) посредством метода `Files.newDirectoryStream(Path dir)`. Этот метод возвращает `DirectoryStream<Path>`, т. е. объект, который мы можем использовать для перебора элементов в каталоге:

```
Path path = Paths.get("D:/learning/books/spring");  
  
try (DirectoryStream<Path> ds = Files.newDirectoryStream(path)) {  
    for (Path file: ds) {  
        System.out.println(file.getFileName());  
    }  
}
```

Если мы хотим дополнить этот фрагмент кода фильтром, у нас есть по крайней мере два варианта решения. Первое решение зависит от еще одной разовидности метода `newDirectoryStream()` — метода `newDirectoryStream(Path dir, String glob)`. Кроме объекта `Path`, этот метод получает фильтр с помощью синтаксиса *glob* (*). Например, мы можем отфильтровать папку `d:/learning/books/spring` для файлов, которые имеют формат PNG, JPG и BMP:

```
try (DirectoryStream<Path> ds =  
    Files.newDirectoryStream(path, "{png,jpg,bmp}")) {
```

```
for (Path file: ds) {
    System.out.println(file.getFileName());
}
}
```

Когда синтаксис `glob` больше не помогает, самое время воспользоваться второй разновидностью метода `newDirectoryStream()`, которая получает фильтр, т. е. `newDirectoryStream(Path dir, DirectoryStream.Filter<? super Path> filter)`. Сначала давайте определим фильтр для файлов размером более 10 Мб:

```
DirectoryStream.Filter<Path> sizeFilter = new DirectoryStream.Filter<>() {
    @Override
    public boolean accept(Path path) throws IOException {
        return (Files.size(path) > 1024 * 1024 * 10);
    }
};
```

Мы также можем сделать это в функциональном стиле:

```
DirectoryStream.Filter<Path> sizeFilter
    = p -> (Files.size(p) > 1024 * 1024 * 10);
```

Теперь мы можем применить этот фильтр следующим образом:

```
try (DirectoryStream<Path> ds = Files.newDirectoryStream(path, sizeFilter)) {
    for (Path file: ds) {
        System.out.println(file.getFileName() + " " +
            Files.readAttributes(file, BasicFileAttributes.class).size()
            + " байт");
    }
}
```

Давайте проверим еще несколько фильтров, которые мы можем использовать с этим техническим приемом.

◆ Пользовательский фильтр для папок:

```
DirectoryStream.Filter<Path> folderFilter
    = new DirectoryStream.Filter<>() {

    @Override
    public boolean accept(Path path) throws IOException {
        return (Files.isDirectory(path, NOFOLLOW_LINKS));
    }
};
```

◆ Пользовательский фильтр для файлов, которые были изменены сегодня:

```
DirectoryStream.Filter<Path> todayFilter
    = new DirectoryStream.Filter<>() {

    @Override
    public boolean accept(Path path) throws IOException {
        FileTime lastModified = Files.readAttributes(path,
            BasicFileAttributes.class).lastModifiedTime();
```

```
LocalDate lastModifiedDate = lastModified.toInstant()
    .atOffset(ZoneOffset.UTC).toLocalDate();
LocalDate todayDate = Instant.now()
    .atOffset(ZoneOffset.UTC).toLocalDate();

return lastModifiedDate.equals(todayDate);
}
};
```

◆ **Пользовательский фильтр для скрытых файлов/папок:**

```
DirectoryStream.Filter<Path> hiddenFilter
    = new DirectoryStream.Filter<>() {

    @Override
    public boolean accept(Path path) throws IOException {
        return (Files.isHidden(path));
    }
};
```

В следующих далее разделах мы рассмотрим разные способы фильтрации файлов.

Фильтрация посредством интерфейса *FilenameFilter*

Функциональный интерфейс `FilenameFilter` также может использоваться для фильтрации файлов из папки. Сначала нам нужно определить фильтр (например, ниже приведен фильтр для файлов типа PDF):

```
String[] files = path.toFile().list(new FilenameFilter() {
    @Override
    public boolean accept(File folder, String fileName) {
        return fileName.endsWith(".pdf");
    }
});
```

Мы можем сделать то же самое в функциональном стиле:

```
FilenameFilter filter = (File folder, String fileName)
    -> fileName.endsWith(".pdf");
```

Давайте сделаем его еще короче:

```
FilenameFilter filter = (f, n) -> n.endsWith(".pdf");
```

Для того чтобы использовать этот фильтр, нам нужно передать его в перегруженный метод `File.list(FilenameFilter filter)` или `File.listFiles(FilenameFilter filter)`:

```
String[] files = path.toFile().list(filter);
```

Массив файлов будет содержать только имена файлов PDF.



Для получения результата в качестве массива `File[]` мы должны вместо `list()` вызвать `listFiles()`.

Фильтрация посредством интерфейса *FileFilter*

`FileFilter` — это еще один функциональный интерфейс, который можно использовать для фильтрации файлов и папок. Например, давайте отфильтруем только папки:

```
File[] folders = path.toFile().listFiles(new FileFilter() {
    @Override
    public boolean accept(File file) {
        return file.isDirectory();
    }
});
```

Мы можем сделать то же самое в функциональном стиле:

```
File[] folders = path.toFile().listFiles((File file)
    -> file.isDirectory());
```

Давайте сделаем этот код еще короче:

```
File[] folders = path.toFile().listFiles(f -> f.isDirectory());
```

Наконец, мы можем сделать это с помощью ссылки:

```
File[] folders = path.toFile().listFiles(File::isDirectory);
```

144. Обнаружение несовпадений между двумя файлами

Решение этой задачи заключается в сравнении содержимого двух файлов (в побайтовом режиме) до тех пор, пока не будет найдено первое несовпадение или не будет достигнут конец файла (EOF).

Рассмотрим четыре текстовых файла (рис. 6.4).

This is a file for testing mismatches between two files! file1.txt	This is a file for testing mismatches between two files! file2.txt	This is a file for testing mismatches between two files! file3.txt	This is a file for testing mismatches between two files. file4.txt

Рис. 6.4

Идентичными являются только первые два файла (`file1.txt` и `file2.txt`). Любое другое сравнение должно выявить наличие хотя бы одного несовпадения.

Один из вариантов решения — использование `MappedByteBuffer`. Это решение является сверхбыстрым и простым в имплементации. Мы просто открываем два файловых канала (по одному для каждого файла) и выполняем байтовое сравнение, пока не найдем первое несовпадение или EOF. Если файлы не имеют одинаковой длины

в байтах, то мы исходим из того, что файлы не являются одинаковыми, и немедленно возвращаемся:

```
private static final int MAP_SIZE = 5242880; // 5 Мбайт в байтах

public static boolean haveMismatches(Path p1, Path p2)
    throws IOException {

    try (FileChannel channel1 = (FileChannel.open(p1,
        EnumSet.of(StandardOpenOption.READ)))) {
        try (FileChannel channel2 = (FileChannel.open(p2,
            EnumSet.of(StandardOpenOption.READ)))) {

            long length1 = channel1.size();
            long length2 = channel2.size();

            if (length1 != length2) {
                return true;
            }

            int position = 0;
            while (position < length1) {
                long remaining = length1 - position;
                int bytestomap = (int) Math.min(MAP_SIZE, remaining);

                MappedByteBuffer mbBuffer1 = channel1.map(
                    MapMode.READ_ONLY, position, bytestomap);
                MappedByteBuffer mbBuffer2 = channel2.map(
                    MapMode.READ_ONLY, position, bytestomap);

                while (mbBuffer1.hasRemaining()) {
                    if (mbBuffer1.get() != mbBuffer2.get()) {
                        return true;
                    }
                }

                position += bytestomap;
            }
        }
    }

    return false;
}
```



JDK 13 подготовил выпуск неволатильных буферов `MappedByteBuffer`. Так что оставайтесь на линии!

Начиная с JDK 12, класс `Files` дополнен новым методом, предназначенным для указания несовпадений между двумя файлами. Этот метод имеет следующую ниже сигнатуру:

```
public static long mismatch(Path path, Path path2) throws IOException
```

Указанный метод находит и возвращает позицию первого несовпадающего байта в содержимом двух файлов. Если несовпадений нет, то возвращается значение `-1`:

```
long mismatches12 = Files.mismatch(file1, file2); // -1
```

```
long mismatches13 = Files.mismatch(file1, file3); // 51
```

```
long mismatches14 = Files.mismatch(file1, file4); // 60
```

145. Кольцевой байтовый буфер

API неблокирующего ввода/вывода NIO.2 идет в комплекте с имплементацией байтового буфера под названием `java.nio.ByteBuffer`. В сущности, он представляет собой массив байтов (`byte[]`), который обернут в набор методов, предназначенных для манипулирования этим массивом (например, `get()`, `put()` и т. д.). *Кольцевой буфер* (циклический буфер, круговой буфер или циклическая очередь) — это буфер фиксированного размера, соединенный встык. На рис. 6.5 показано, как выглядит круговая очередь.

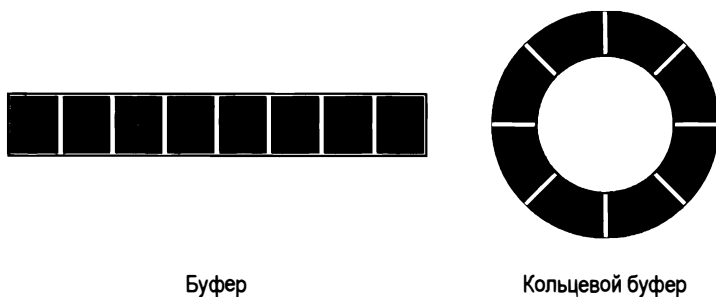
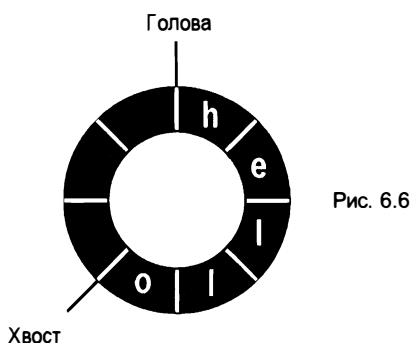


Рис. 6.5

Кольцевой буфер опирается на предварительно выделенный массив (предварительно выделенной емкости), но некоторые имплементации могут также потребовать возможности изменения его размера. Элементы записываются/добавляются в конец (*хвост*) и удаляются/считываются спереди (из *головы*); это видно на рис. 6.6.



Для главных операций, т. е. чтения (get) и записи (put), кольцевой буфер поддерживает указатель (указатель чтения и указатель записи). Оба указателя обернуты вокруг емкости буфера. Мы можем в любое время узнать, сколько элементов имеется для чтения и сколько свободных слотов можно записать. Эта операция выполняется за $O(1)$.

Кольцевой байтовый буфер может иметь значения типа `char` или какой-либо другого типа. Это именно то, что мы хотим здесь имплементировать. Мы можем начать с написания заготовки нашей имплементации:

```
public class CircularByteBuffer {
    private int capacity;
    private byte[] buffer;
    private int readPointer;
    private int writePointer;
    private int available;

    CircularByteBuffer(int capacity) {
        this.capacity = capacity;
        buffer = new byte[capacity];
    }

    public synchronized int available() {
        return available;
    }

    public synchronized int capacity() {
        return capacity;
    }

    public synchronized int slots() {
        return capacity - available;
    }

    public synchronized void clear() {
        readPointer = 0;
        writePointer = 0;
        available = 0;
    }
    ...
}
```

Теперь давайте сосредоточимся на вводе (записи) новых байтов и чтении (получении) существующих байтов. Например, кольцевой байтовый буфер емкостью 8 может быть представлен так, как показано на рис. 6.7.

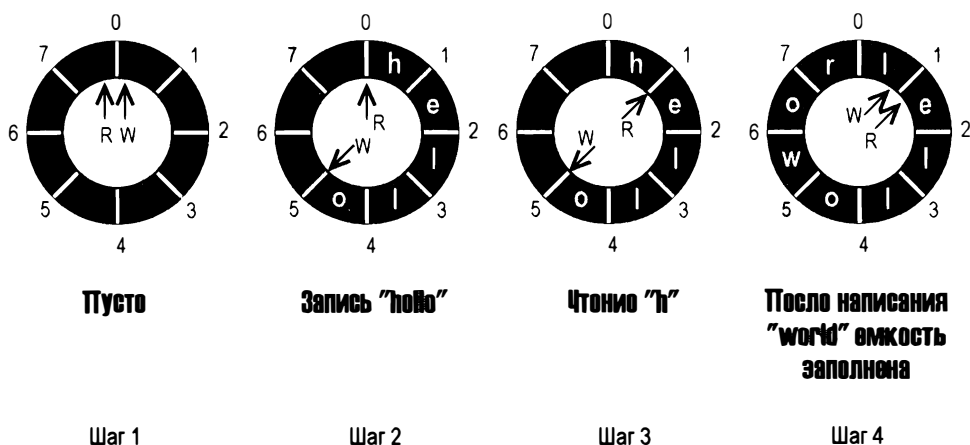


Рис. 6.7

Давайте посмотрим, что происходит на каждом шаге:

Кольцевой байтовый буфер пуст, и оба указателя указывают на слот 0 (первый слот).

Мы помещаем 5 байт, соответствующих слову `hello`, в буфер. Указатель `readPointer` остается в том же положении, в то время как указатель `writePointer` указывает на слот 5.

Мы получаем байты, соответствующие букве `h`, поэтому указатель `readPointer` перемещается в слот 1.

Наконец, мы попытаемся поместить байты слова `world` в буфер. Это слово состоит из 5 байт, но у нас есть только четыре свободных слота, после чего мы достигнем предела буферной емкости. Это означает, что мы можем записать только те байты, которые соответствуют `worl`.

Теперь давайте рассмотрим сценарий на рис. 6.8.

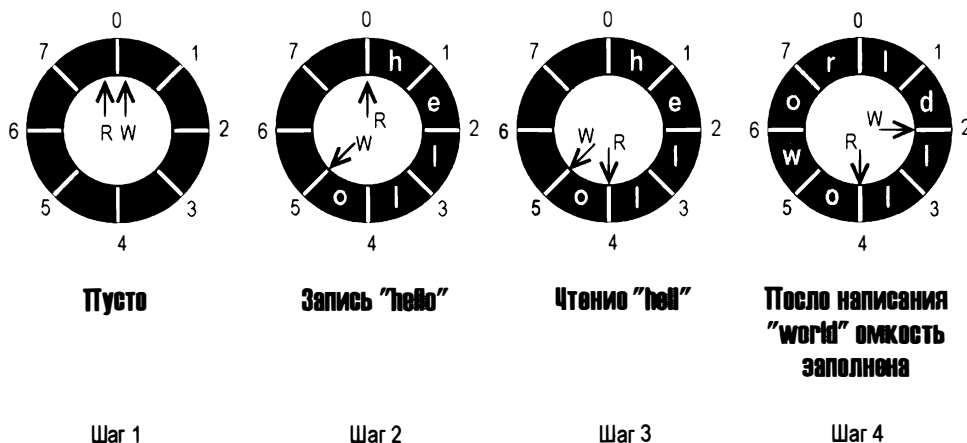


Рис 6 8

Слева направо следующие шаги:

1. Первые два шага те же, что и в предыдущем сценарии.
2. Мы получаем байты слова `hell`. Это переместит указатель `readPointer` в позицию 4.
3. Наконец, мы помещаем байты слова `world` в буфер. На этот раз это слово помещается в буфер, и `writePointer` перемещается в слот 2.

Основываясь на этой процедуре, мы можем легко имплементировать метод, который помещает 1 байт в буфер, и еще один метод, который получает 1 байт из буфера, следующим образом:

```
public synchronized boolean put(int value) {
    if (available == capacity) {
        return false;
    }

    buffer[writePointer] = (byte) value;
    writePointer = (writePointer + 1) % capacity;
    available++;

    return true;
}

public synchronized int get() {
    if (available == 0) {
        return -1;
    }

    byte value = buffer[readPointer];
    readPointer = (readPointer + 1) % capacity;
    available--;

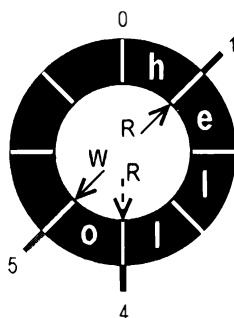
    return value;
}
```

Если мы проверим API неблокирующего ввода/вывода NIO.2 `ByteBuffer`, то заметим, что он предоставляет несколько разновидностей методов `get()` и `put()`. Например, мы должны уметь передавать массив `byte[]` методу `get()`, и этот метод должен копировать диапазон элементов из буфера в этот массив `byte[]`. Элементы считываются из буфера, начиная с текущей точки чтения, и записываются в заданном массиве `byte[]`, начиная с указанного смещения `offset`.

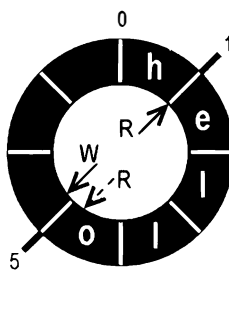
Схема на рис. 6.9 показывает случай, когда указатель `writePointer` больше указателя `readPointer`.

В левой части рисунка показано, что мы читаем 3 байта. При этом указатель `readPointer` перемещается из своего начального слота 1 в слот 4. Справа на рис. 6.9 показано, что мы читаем 4 байта (или более 4). Поскольку доступны только 4 байта,

указатель `readPointer` перемещается из своего начального слота в тот же слот, что и указатель `writePointer` (слот 5).



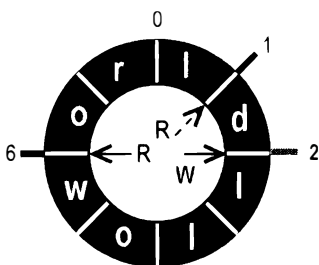
Чтоnio 3 байт



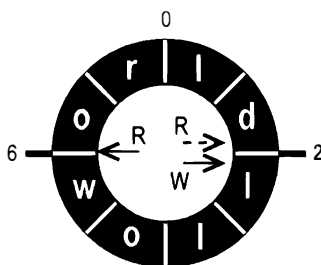
Чтоnio 4 байт (или больше)

Рис. 6.9

Теперь давайте проанализируем случай, когда указатель `writePointer` меньше указателя `readPointer` (рис. 6.10).



Чтоnio 3 байт



Чтоnio 4 байт (или больше)

Рис. 6.10

На рис. 6.10 слева показано, что мы читаем 3 байта. Здесь указатель `readPointer` перемещается из его начального слота 6 в слот 1. Справа показано, что мы читаем 4 байта (или более 4). Это перемещает указатель `readPointer` из его начального слота 6 в слот 2 (тот же слот, что и у указателя `writePointer`).

Теперь, когда нам известны эти два варианта использования, мы можем написать метод `get()`, чтобы скопировать диапазон байтов из буфера в заданный массив `byte[]`, следующим образом (этот метод пытается прочитать `len` байт из буфера и записать их в заданный массив `byte[]`, начиная с заданного смещения `offset`):

```
public synchronized int get(byte[] dest, int offset, int len) {
    if (available == 0) {
        return 0;
    }
}
```

```

int maxPointer = capacity;

if (readPointer < writePointer) {
    maxPointer = writePointer;
}

int countBytes = Math.min(maxPointer - readPointer, len);
System.arraycopy(buffer, readPointer, dest, offset, countBytes);
readPointer = readPointer + countBytes;

if (readPointer == capacity) {
    int remainingBytes = Math.min(len - countBytes, writePointer);

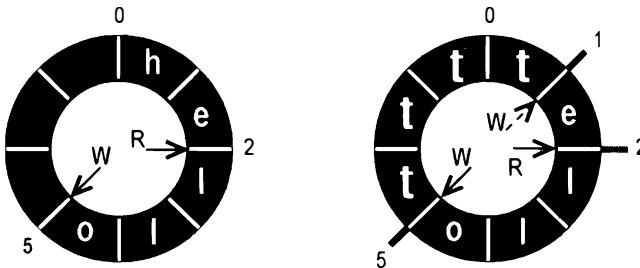
    if (remainingBytes > 0) {
        System.arraycopy(buffer, 0, dest,
            offset + countBytes, remainingBytes);
        readPointer = remainingBytes;
        countBytes = countBytes + remainingBytes;
    } else {
        readPointer = 0;
    }
}

available = available - countBytes;

return countBytes;
}

```

Теперь давайте сосредоточимся на том, чтобы поместить заданный массив `byte[]` в буфер. Элементы считываются из заданного массива `byte[]`, начиная с указанного смещения, и записываются в буфер, начиная с текущего указателя `writePointer`. На рис. 6.11 показан случай, когда указатель `writePointer` больше указателя `readPointer`.



Начальное состояние

Чтение 4 байт, "tttt"

Рис. 6.11

На рис. 6.11 слева мы имеем начальное состояние буфера. Итак, указатель `readPointer` указывает на слот 2, а указатель `writePointer` — на слот 5. После записи 4 байт (на рис. 6.11 справа) мы видим, что указатель `readPointer` не был затронут, и указатель `writePointer` указывает на слот 1.

Другой вариант использования исходит из того, что указатель `readPointer` больше указателя `writePointer` (рис. 6.12).

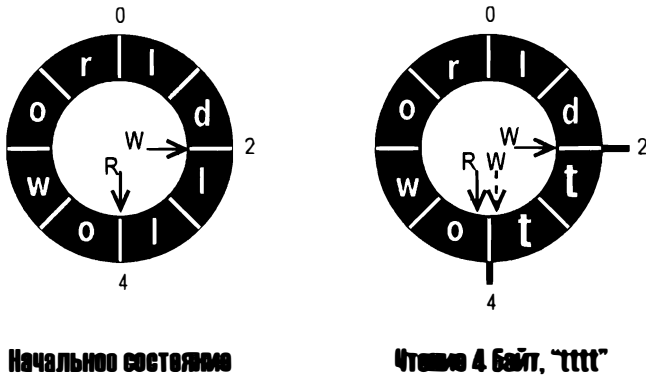


Рис. 6.12

На рис. 6.12 слева мы имеем начальное состояние буфера. Указатель `readPointer` указывает на слот 4, а указатель `writePointer` — на слот 2. После записи 4 байт (на рис. 6.12 справа) мы можем видеть, что указатель `readPointer` не был затронут и указатель `writePointer` указывает на слот 4. Обратите внимание, что только 2 байта были успешно записаны. Это произошло потому, что мы достигли максимальной емкости буфера перед записью всех 4 байт.

Теперь, имея в арсенале эти два варианта использования, мы можем написать метод `put()`, чтобы скопировать диапазон байтов из заданного массива `byte[]` в буфер, следующим образом (этот метод пытается прочитать `len` байт из заданного массива `byte[]`, начиная с заданного смещения `offset`, и записать их в буфер, начиная с текущего указателя `writePointer`):

```
public synchronized int put(byte[] source, int offset, int len) {
    if (available == capacity) {
        return 0;
    }

    int maxPointer = capacity;

    if (writePointer < readPointer) {
        maxPointer = readPointer;
    }
}
```

```
int countBytes = Math.min(maxPointer - writePointer, len);
System.arraycopy(source, offset, buffer, writePointer, countBytes);
writePointer = writePointer + countBytes;

if (writePointer == capacity) {
    int remainingBytes = Math.min(len - countBytes, readPointer);

    if (remainingBytes > 0) {
        System.arraycopy(source, offset + countBytes, buffer, 0, remainingBytes);
        writePointer = remainingBytes;
        countBytes = countBytes + remainingBytes;
    } else {
        writePointer = 0;
    }
}

available = available + countBytes;

return countBytes;
}
```

Как мы уже упоминали ранее, иногда нам нужно изменить размер буфера. Например, мы можем захотеть удвоить его размер, просто вызвав метод `resize()`. В сущности, это означает копирование всех имеющихся байтов (элементов) в новый буфер с двойной емкостью:

```
public synchronized void resize() {
    byte[] newBuffer = new byte[capacity * 2];

    if (readPointer < writePointer) {
        System.arraycopy(buffer, readPointer, newBuffer, 0, available);
    } else {
        int bytesToCopy = capacity - readPointer;
        System.arraycopy(buffer, readPointer, newBuffer, 0, bytesToCopy);
        System.arraycopy(buffer, 0, newBuffer, bytesToCopy, writePointer);
    }

    buffer = newBuffer;
    capacity = buffer.length;
    readPointer = 0;
    writePointer = available;
}
```

Обратитесь к исходному коду, прилагаемому к этой книге, чтобы увидеть, как он работает в полном объеме.

146. Лексемизация файлов

Содержимое файла не всегда принимается таким, чтобы его можно было обработать немедленно, и до начала обработки требуются некоторые дополнительные подготовительные шаги. В типичной ситуации нам нужно лексемизировать (токенизировать, разбить на токены) файл и извлечь информацию из разных структур данных (массивов, списков, отображений и т. д.).

Например, давайте рассмотрим файл `clothes.txt`:

```
Path path = Paths.get("clothes.txt");
```

Его содержимым являются следующие данные:

```
Top|white\10/XXL&Swimsuit|black\5/L
```

```
Coat|red\11/M&Golden Jacket|yellow\12/XLDenim|Blue\22/M
```

Этот файл содержит несколько предметов одежды и их детали, разделенные символом `&`. Один предмет представлен следующим образом:

название предмета | цвет \ количество предметов в наличии / размер

Здесь у нас несколько разделителей (`&`, `|`, `\`, `/`) и очень специфический формат.

Теперь давайте рассмотрим несколько вариантов решения по извлечению и лексемизации информации из этого файла в качестве списка. Мы соберем эту информацию в служебном классе `FileTokenizer`.

Одно из решений по извлечению предметов в список опирается на метод `String.split()`. Мы должны прочитать файл построчно и применить метод `String.split()` к каждой строке файла. Результат лексемизации каждой строки файла собирается в список `List` посредством метода `List.addAll()`:

```
public static List<String> get(Path path,
    Charset cs, String delimiter) throws IOException {

    String delimiterStr = Pattern.quote(delimiter);
    List<String> content = new ArrayList<>();

    try (BufferedReader br = Files.newBufferedReader(path, cs)) {

        String line;
        while ((line = br.readLine()) != null) {
            String[] values = line.split(delimiterStr);
            content.addAll(Arrays.asList(values));
        }
    }

    return content;
}
```


Вызов этого метода с разделителем `&#x26amp;` приведет к следующему результату:

```
[Top|white\10/XXL, Swimsuit|black\5/L, Coat|red\11/M, Golden  
Jacket|yellow\12/XL, Denim|Blue\22/M]
```

Еще одна изюминка предыдущего решения может опереться на метод `Collectors.toList()` вместо метода `Arrays.asList()`:

```
public static List<String> get(Path path,  
    Charset cs, String delimiter) throws IOException {  
  
    String delimiterStr = Pattern.quote(delimiter);  
    List<String> content = new ArrayList<>();  
  
    try (BufferedReader br = Files.newBufferedReader(path, cs)) {  
        String line;  
        while ((line = br.readLine()) != null) {  
            content.addAll(Stream.of(line.split(delimiterStr))  
                .collect(Collectors.toList()));  
        }  
    }  
  
    return content;  
}
```

В качестве альтернативы мы можем обрабатывать содержимое в ленивой манере посредством метода `Files.lines()`:

```
public static List<String> get(Path path,  
    Charset cs, String delimiter) throws IOException {  
  
    try (Stream<String> lines = Files.lines(path, cs)) {  
  
        return lines.map(l -> l.split(Pattern.quote(delimiter)))  
            .flatMap(Arrays::stream)  
            .collect(Collectors.toList());  
    }  
}
```

В случае относительно малых файлов мы можем загрузить файл в память и обработать соответствующим образом:

```
Files.readAllLines(path, cs).stream()  
    .map(l -> l.split(Pattern.quote(delimiter)))  
    .flatMap(Arrays::stream)  
    .collect(Collectors.toList());
```

Еще один вариант решения может опираться на метод `Pattern.splitAsStream()` JDK 8. Этот метод создает поток из заданной входной последовательности. Для разно-

образия на этот раз давайте соберем результирующий список посредством метода `Collectors.joining(";")`:

```
public static List<String> get(Path path,
    Charset cs, String delimiter) throws IOException {

    Pattern pattern = Pattern.compile(Pattern.quote(delimiter));
    List<String> content = new ArrayList<>();

    try (BufferedReader br = Files.newBufferedReader(path, cs)) {
        String line;
        while ((line = br.readLine()) != null) {
            content.add(pattern.splitAsStream(line)
                .collect(Collectors.joining(";")));
        }
    }

    return content;
}
```

Давайте вызовем этот метод с разделителем `&` :

```
List<String> tokens = FileTokenizer.get(
    path, StandardCharsets.UTF_8, "&amp;");
```

В результате мы получаем следующее:

```
[Top|white\10/XXL;Swimsuit|black\5/L, Coat|red\11/M;Golden
Jacket|yellow\12/XL, Denim|Blue\22/M]
```

До сих пор представленные варианты решения получали список предметов, применяя один разделитель. Но иногда нам нужно применить более одного разделителя. Например, допустим, что мы хотим получить следующий результат (список):

```
[Top, white, 10, XXL, Swimsuit, black, 5, L, Coat, red, 11, M, Golden
Jacket, yellow, 12, XL, Denim, Blue, 22, M]
```

Для того чтобы получить такой список, мы должны применить несколько разделителей (`&`, `|`, `\` и `/`). Это можно сделать посредством метода `String.split()` и передачи ему регулярного выражения, основанного на логическом операторе ИЛИ (`x|y`):

```
public static List<String> getWithMultipleDelimiters(
    Path path, Charset cs, String...delimiters) throws IOException {

    String[] escapedDelimiters = new String[delimiters.length];
    Arrays.setAll(escapedDelimiters, t -> Pattern.quote(delimiters[t]));
    String delimiterStr = String.join("|", escapedDelimiters);

    List<String> content = new ArrayList<>();

    try (BufferedReader br = Files.newBufferedReader(path, cs)) {
        String line;
```

```
while ((line = br.readLine()) != null) {
    String[] values = line.split(delimiterStr);
    content.addAll(Arrays.asList(values));
}

return content;
}
```

Давайте вызовем этот метод с нашими разделителями (&, |, \ и /), чтобы получить требуемый результат:

```
List<String> tokens = FileTokenizer.getWithMultipleDelimiters(
    path, StandardCharsets.UTF_8,
    new String[] {"&", "|", "\\ ", "/"});
```

ОК, пока все идет хорошо! Все эти варианты решения основаны на методах `String.split()` и `Pattern.splitAsStream()`. Еще одно множество решений может опираться на класс `StringTokenizer` (он не отличается высокой производительностью, поэтому используйте его осторожно). Этот класс может применять разделитель (или несколько разделителей) к заданной строке и выставляет наружу два главных метода для управления ею, а именно методы `hasMoreElements()` и `nextToken()`:

```
public static List<String> get(Path path,
    Charset cs, String delimiter) throws IOException {

    StringTokenizer st;
    List<String> content = new ArrayList<>();

    try (BufferedReader br = Files.newBufferedReader(path, cs)) {
        String line;
        while ((line = br.readLine()) != null) {
            st = new StringTokenizer(line, delimiter);
            while (st.hasMoreElements()) {
                content.add(st.nextToken());
            }
        }
    }

    return content;
}
```

Его можно также использовать в сочетании с коллекторами:

```
public static List<String> get(Path path,
    Charset cs, String delimiter) throws IOException {

    List<String> content = new ArrayList<>();
```

```
try (BufferedReader br = Files.newBufferedReader(path, cs)) {
    String line;
    while ((line = br.readLine()) != null) {
        content.addAll(Collections.list(
            new StringTokenizer(line, delimiter)).stream()
                .map(t -> (String) t)
                .collect(Collectors.toList()));
    }
}

return content;
}
```

Можно использовать несколько разделителей, если мы разделяем их с помощью `//`:

```
public static List<String> getWithMultipleDelimiters(
    Path path, Charset cs, String...delimiters) throws IOException {

    String delimiterStr = String.join("//", delimiters);
    StringTokenizer st;
    List<String> content = new ArrayList<>();

    try (BufferedReader br = Files.newBufferedReader(path, cs)) {
        String line;
        while ((line = br.readLine()) != null) {
            st = new StringTokenizer(line, delimiterStr);
            while (st.hasMoreElements()) {
                content.add(st.nextToken());
            }
        }
    }

    return content;
}
```



Для получения более высокой производительности и поддержки со стороны регулярных выражений (т. е. высокой гибкости) рекомендуется опираться на метод `String.split()` вместо `StringTokenizer`. Из этой же категории рассмотрите также *разд. 148 "Работа с классом `Scanner`" далее в этой главе*.

147. Запись форматированного вывода непосредственно в файл

Предположим, что у нас есть 10 чисел (целых и двойной точности) и мы хотим, чтобы они были хорошо отформатированы (имели отступ, выравнивание и ряд десятичных знаков, которые поддерживают удобочитаемость и полезность) в файле.

В нашей первой попытке мы записали их в файл вот так (форматирование не применялось):

```
Path path = Paths.get("noformatter.txt");

try (BufferedWriter bw = Files.newBufferedWriter(path,
    StandardCharsets.UTF_8, StandardOpenOption.CREATE,
    StandardOpenOption.WRITE)) {

    for (int i = 0; i < 10; i++) {
        bw.write("| " + intValues[i] + " | " + doubleValues[i] + " | ");
        bw.newLine();
    }
}
```

Результат приведенного выше исходного кода аналогичен тому, что показано на рис. 6.13 слева.

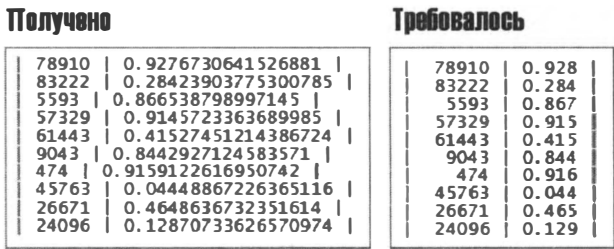


Рис. 6.13

Однако мы хотим получить результат, который показан на рис. 6.13 справа. Для того чтобы решить эту задачу, нам необходимо использовать метод `String.format()`. Он позволяет задавать правила форматирования в качестве строкового значения, которое соответствует следующему шаблону:

```
%[флаги][ширина][.точность]символ-конверсии
```

Посмотрим, что представляет собой каждый компонент этого шаблона.

- ◆ Компонент `[флаги]` является необязательным и состоит из стандартных подходов к модифицированию выходных данных. Часто они используются для форматирования целых чисел и чисел с плавающей точкой.
- ◆ Компонент `[ширина]` является необязательным и задает ширину поля для наших выходных данных (минимальное число символов, записываемых в выходные данные).
- ◆ Компонент `[.точность]` является необязательным и определяет число цифр точности для значений с плавающей точкой (или длину подстроки для извлечения из значения типа `String`).

◆ Компонент *символ-конверсии* является обязательным и сообщает нам о том, как аргумент будет отформатирован. Наиболее часто используемыми символами конверсии являются следующие:

- `s` — для форматирования строк;
- `d` — для форматирования десятичных целых чисел;
- `f` — для форматирования чисел с плавающей точкой;
- `t` — для форматирования значений даты/времени.



В качестве разделителя строк можно использовать `%n`.

С этим знанием правил форматирования мы можем получить то, что хотим, следующим образом (`%6s` используется для целых чисел, в то время как `%.3f` применяется для чисел двойной точности):

```
Path path = Paths.get("withformatter.txt");

try (BufferedWriter bw = Files.newBufferedWriter(path,
    StandardCharsets.UTF_8, StandardOpenOption.CREATE,
    StandardOpenOption.WRITE)) {

    for (int i = 0; i < 10; i++) {
        bw.write(String.format("| %6s | %.3f |",
            intValues[i], doubleValues[i]));
        bw.newLine();
    }
}
```

Еще один вариант решения может быть предоставлен посредством класса `Formatter`. Он предназначен для форматирования строк и использует те же правила форматирования, что и метод `String.format()`. В классе имеется метод `format()`, который мы можем использовать для того, чтобы переписать предыдущий фрагмент кода:

```
Path path = Paths.get("withformatter.txt");

try (Formatter output = new Formatter(path.toFile())) {
    for (int i = 0; i < 10; i++) {
        output.format("| %6s | %.3f |%n", intValues[i], doubleValues[i]);
    }
}
```

Как насчет форматирования только целых чисел (рис. 6.14)?

```

78,910 bytes
83,222 bytes
5,593 bytes
57,329 bytes
61,443 bytes
9,043 bytes
474 bytes
45,763 bytes
26,671 bytes
24,096 bytes

```

Рис. 6.14

Что ж, мы можем получить его, применив `DecimalFormat` и строковый формater, следующим образом:

```

Path path = Paths.get("withformatter.txt");
DecimalFormat formatter = new DecimalFormat("###,### байт");

try (Formatter output = new Formatter(path.toFile())) {
    for (int i = 0; i < 10; i++) {
        output.format("%12s%n", formatter.format(intValues[i]));
    }
}

```

148. Работа с классом *Scanner*

Класс `Scanner` выставляет наружу API для проведения лексического разбора текста из строковых значений, файлов, консоли и т. д. *Лексический разбор* — это процесс разбивки входных данных на лексемы (токены) и возврата их по мере необходимости (например, разбивки на целые числа, числа с плавающей точкой, числа двойной точности и т. д.). По умолчанию `Scanner` разбирает входные данные, используя пробел (разделитель, заданный по умолчанию) и предоставляет лексемы/токены посредством связки методов `nextFoo()` (например, `next()`, `nextLine()`, `nextInt()`, `nextDouble()` и т. д.).



Из той же категории задач рассмотрите *разд. 146 "Лексемизация файлов"*.

Например, допустим, что у нас есть файл (`doubles.txt`), который содержит числа двойной точности, разделенные пробелами (рис. 6.15).

```

double.txt
23.4556 1.23 4.55 2.33
5.663 956.34343 23.2333
0.3434 0.788

```

Рис 6.15

Если мы хотим получить этот текст в виде чисел двойной точности, то можем прочитать его и опереться на фрагмент спагетти-кода, который будет его лексемизиро-

вать и конвертировать в числа двойной точности. В качестве альтернативы мы можем опереться на класс `Scanner` и его метод `nextDouble()`, как показано ниже:

```
try (Scanner scanDoubles = new Scanner(
    Path.of("doubles.txt"), StandardCharsets.UTF_8)) {

    while (scanDoubles.hasNextDouble()) {
        double number = scanDoubles.nextDouble();
        System.out.println(number);
    }
}
```

Результат приведенного выше фрагмента кода выглядит следующим образом:

```
23.4556
1.23
...
```

Однако файл может содержать смешанную информацию разных типов. Например, файл (`people.txt`) на рис. 6.16 содержит строковые значения и целые числа, разделенные разными разделителями (запятая и точка с запятой).

people.txt

Matt,Kyle,23,San Francisco; Dare1,Der,50,New York;Sandra,Hui,40,Dallas; Leonard,Vurt,43,Bucharest;Mark,Seil,19,Texas;Ulm,Bar,43,Kansas

Рис 6.16

Класс `Scanner` предоставляет метод, именуемый `useDelimiter()`. Этот метод берет аргумент типа `String` или `Pattern`, для того чтобы указать разделитель (разделители), который должен использоваться в качестве регулярного выражения:

```
try (Scanner scanPeople = new Scanner(Path.of("people.txt"),
    StandardCharsets.UTF_8).useDelimiter(";|,")) {

    while (scanPeople.hasNextLine()) {
        System.out.println("Name: " + scanPeople.next().trim());
        System.out.println("Surname: " + scanPeople.next());
        System.out.println("Age: " + scanPeople.nextInt());
        System.out.println("City: " + scanPeople.next());
    }
}
```

Результат использования этого метода выглядит следующим образом:

```
Name: Matt
Surname: Kyle
Age: 23
City: San Francisco
...
```


Начиная с JDK 9, класс `Scanner` предоставляет новый метод под названием `tokens()`. Этот метод возвращает поток лексем/токенов, разделенных разделителями, из сканера. Например, мы можем использовать его для разбора файла `people.txt` и печати его в консоли, как показано ниже:

```
try (Scanner scanPeople = new Scanner(Path.of("people.txt"),
    StandardCharsets.UTF_8).useDelimiter(";|,")) {

    scanPeople.tokens().forEach(t -> System.out.println(t.trim()));
}
```

Результат использования указанного выше метода выглядит следующим образом:

```
Matt
Kyle
23
San Francisco
...
```

В качестве альтернативы мы можем соединить лексемы/токены пробелом:

```
try (Scanner scanPeople = new Scanner(Path.of("people.txt"),
    StandardCharsets.UTF_8).useDelimiter(";|,")) {

    String result = scanPeople.tokens()
        .map(t -> t.trim())
        .collect(Collectors.joining(" "));
}
```



В разд. 140 "Поиск в больших файлах" ранее в этой главе приведен пример использования данного метода для поиска того или иного фрагмента текста в файле.

Результат использования указанного выше метода выглядит следующим образом:

```
Matt Kyle 23 San Francisco Darel Der 50 New York ...
```

С точки зрения методов `tokens()`, JDK 9 также идет в комплекте с методом `findAll()`. Этот метод очень удобен для поиска всех лексем/токенов, которые относятся к некоторому регулярному выражению (предоставляемому в качестве экземпляра `String` или `Pattern`). Метод возвращает поток `Stream<MatchResult>` и может использоваться следующим образом:

```
try (Scanner sc = new Scanner(Path.of("people.txt"))) {
    Pattern pattern = Pattern.compile("4[0-9]");

    List<String> ages = sc.findAll(pattern)
        .map(MatchResult::group)
        .collect(Collectors.toList());

    System.out.println("Ages: " + ages);
}
```

В приведенном выше фрагменте кода выбираются все лексемы/токены, представляющие возраст от 40 до 49 лет, т. е. 40, 43 и 43.

Класс `Scanner` является удобным подходом для использования, если мы хотим лексемизировать входные данные, предоставляемые из консоли:

```
Scanner scanConsole = new Scanner(System.in);

String name = scanConsole.nextLine();
String surname = scanConsole.nextLine();
int age = scanConsole.nextInt();
// int не может включать "\n", поэтому нам нужна
// следующая ниже строка, которая потребит "\n"
scanConsole.nextLine();
String city = scanConsole.nextLine();
```



Обратите внимание, что для числовых входных данных (читаемых посредством методов `nextInt()`, `nextFloat()` и т. д.) нам также нужно использовать символ новой строки (вставляется, когда мы нажимаем клавишу <Enter>). Сканер не будет извлекать этот символ при разборе числа, и поэтому он появится в следующей лексеме. Если мы не потребим его путем добавления строки кода `nextLine()`, то с этого момента входные данные будут невыровненными, что станет причиной исключения `InputMismatchException` или преждевременного завершения работы.

Конструкторы класса `Scanner`, поддерживающие наборы символов, были введены в JDK 10.

Давайте посмотрим на разницу между классами `Scanner` и `BufferedReader`.

Scanner против BufferedReader

Итак, что нам следует использовать: `Scanner` или же `BufferedReader`? Скажем так, если нам нужно разобрать файл, то `Scanner` является лучшим способом; в противном случае больше подходит `BufferedReader`. Их прямой сравнительный анализ показывает следующее.

- ◆ Класс `BufferedReader` работает быстрее класса `Scanner`, т. к. не выполняет никаких операций разбора.
- ◆ Класс `BufferedReader` превосходит в том, что касается чтения, в то время как класс `Scanner` имеет преимущества в разборе.
- ◆ По умолчанию класс `BufferedReader` использует буфер 8 Кбайт, в то время как классу `Scanner` нужен буфер 1 Кбайт.
- ◆ Класс `BufferedReader` хорошо подходит для чтения длинных строк, в то время как класс `Scanner` эффективнее для коротких входов.
- ◆ Класс `BufferedReader` синхронизирован, а класс `Scanner` — нет.

- ◆ Класс `Scanner` может использовать класс `BufferedReader`, в то время как обратное невозможно. Это показано в следующем ниже фрагменте кода:

```
try (Scanner scanDoubles = new Scanner(Files.newBufferedReader(
    Path.of("doubles.txt"), StandardCharsets.UTF_8))) {
    ...
}
```

Резюме

Мы подошли к концу главы, в которой познакомились с разными специфическими задачами ввода/вывода. Мы рассмотрели много тем — от манипулирования путями, их посещения и наблюдения за ними до потоковой передачи файлов и эффективных способов чтения/записи текстовых и двоичных файлов.

Вы можете скачать приложения из этой главы, чтобы увидеть результаты и получить дополнительные детали, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

7

Классы, интерфейсы, конструкторы, методы и поля в API рефлексии Java

Эта глава содержит 17 задач, связанных с API рефлексии Java. Здесь представлен полный обзор API рефлексии Java от классических тем, таких как проверка и создание экземпляров артефактов Java (например, модулей, пакетов, классов, интерфейсов, надклассов, конструкторов, методов, аннотаций и массивов) до синтетических и мостовых конструкций или управления гнездовым доступом (JDK 11).

К концу этой главы у API рефлексии Java не останется никаких секретов и вы будете готовы показать своим коллегам все, что способна делать рефлексия.

Задачи

Используйте следующие задачи для проверки вашего умения программировать API рефлексии Java. Перед тем как обратиться к решениям и скачать примеры программ, настоятельно рекомендуется изучить каждую задачу.

149. **Инспектирование пакетов.** Написать несколько примеров инспектирования пакетов Java (например, имен, списка классов и т. д.).
150. **Инспектирование классов и надклассов.** Написать несколько примеров инспектирования классов и надклассов (например, получения специального класса `Class` посредством имени класса, модификаторы, имплементированные интерфейсы, конструкторы, методы и поля).
151. **Создание экземпляра класса посредством отрефлексированного конструктора.** Написать программу, которая создает экземпляры посредством рефлексии.

152. **Извлечение аннотации типа получателя.** Написать программу, которая извлекает аннотацию типа получателя.
153. **Получение синтетических и мостовых конструкций.** Написать программу, которая получает синтетические и мостовые конструкции посредством рефлексии.
154. **Проверка переменного числа аргументов.** Написать программу, которая проверяет, получает ли метод переменное число аргументов.
155. **Проверка методов по умолчанию.** Написать программу, которая проверяет, является ли метод установленным по умолчанию.
156. **Управление гнездовым доступом посредством рефлексии.** Написать программу, которая обеспечивает гнездовой доступ к конструкциям посредством рефлексии.
157. **Рефлексия для геттеров и сеттеров.** Написать несколько примеров, которые вызывают геттеры и сеттеры посредством рефлексии. Кроме того, написать программу, которая генерирует геттеры и сеттеры с помощью рефлексии.
158. **Рефлекси́рование аннотаций.** Написать несколько примеров получения разных видов аннотаций посредством рефлексии.
159. **Вызов экземплярного метода.** Написать программу, которая вызывает экземплярный метод посредством рефлексии.
160. **Получение статических методов.** Написать программу, которая группирует статические методы заданного класса и вызывает один из них посредством рефлексии.
161. **Получение обобщенных методов, полей и исключений.** Написать программу, которая получает обобщенные типы методов, полей и исключений посредством рефлексии.
162. **Получение публичных и приватных полей.** Написать программу, которая получает публичные и приватные поля заданного класса посредством рефлексии.
163. **Работа с массивами.** Написать несколько примеров работы с массивами посредством рефлексии.
164. **Инспектирование модулей.** Написать несколько примеров обследования модулей Java 9 с помощью рефлексии.
165. **Динамические посредники.** Написать программу, которая использует динамических посредников (прокси) для подсчета числа вызовов методов заданных интерфейсов.

Решения

В следующих разделах описаны решения перечисленных выше задач. Помните, что обычно не существует одного единственно правильного варианта решения той или иной задачи. Кроме того, имейте в виду, что приведенные здесь объяснения включают только самые интересные и важные детали, необходимые для решения. Вы можете скачать примеры решений, чтобы увидеть дополнительные детали и поэкспериментировать с программами, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

149. Инспектирование пакетов

Класс `java.lang.Package` находится в центре нашего внимания, когда нам нужно получить информацию о конкретном пакете. С помощью этого класса мы можем узнать имя пакета, поставщика, который имплементировал этот пакет, его имя, версию пакета и т. д.

Этот класс обычно используется для поиска имени пакета, содержащего тот или иной класс. Например, имя пакета класса `Integer` можно легко получить следующим образом:

```
Class clazz = Class.forName("java.lang.Integer");
Package packageOfClazz = clazz.getPackage();

// java.lang
String packageNameOfClazz = packageOfClazz.getName();

Теперь найдем имя пакета класса File:

File file = new File(".");
Package packageOfFile = file.getClass().getPackage();

// java.io
String packageNameOfFile = packageOfFile.getName();
```



Если мы пытаемся найти имя пакета текущего класса, то можем в этом полагаться на метод `this.getClass().getPackage().getName()`. Он работает в нестатическом контексте.

Но если мы хотим лишь быстро перечислить все пакеты текущего загрузчика классов, то можем полагаться на метод `getPackages()`, как показано ниже:

```
Package[] packages = Package.getPackages();
```

Основываясь на методе `getPackages()`, мы можем перечислить все пакеты, определенные загрузчиком классов, а также его предков, которые начинаются с заданного префикса, следующим образом:

```
public static List<String> fetchPackagesByPrefix(String prefix) {
    return Arrays.stream(Package.getPackages())
```

```
.map(Package::getName)
.filter(n -> n.startsWith(prefix))
.collect(Collectors.toList());
}
```

Если этот метод располагается в служебном классе с именем `Packages`, то мы можем вызвать его следующим образом:

```
List<String> packagesSamePrefix = Packages.fetchPackagesByPrefix("java.util");
```

Вы увидите результат, подобный следующему:

```
java.util.function, java.util.jar, java.util.concurrent.locks,
java.util.spi, java.util.logging, ...
```

Иногда мы просто хотим перечислить все классы пакета в системном загрузчике классов. Давайте посмотрим, как это можно сделать.

Получение классов пакета

Например, мы можем захотеть перечислить классы одного из пакетов текущего приложения (скажем, пакета `modern.challenge`) или классы одного из пакетов из наших библиотек времени компиляции (например, `commonslang-2.4.jar`).

Классы обернуты в пакеты, которые могут быть архивированными в файлы JAR, хотя это и не обязательно. Для того чтобы охватить оба случая, нам нужно выяснить, располагается ли данный пакет в файле JAR или нет. Мы можем сделать это, загрузив ресурс посредством метода `ClassLoader.getSystemClassLoader().getResource(package_path)` и проверив возвращенный URL-адрес ресурса. Если пакет не располагается в файле JAR, то ресурсом будет URL-адрес, начинающийся с `file:`, как в следующем ниже примере (мы используем `modern.challenge`):

```
file:/D:/Java%20Modern%20Challenge/Code/Chapter%207/Inspect%20packages
/build/classes/modern/challenge
```

Но если пакет находится внутри файла JAR (например, `org.apache.commons.lang3.builder`), то URL-адрес будет начинаться с `jar:`, как в следующем примере:

```
jar:file:/D:/.../commons-lang3-3.9.jar!/org/apache/commons/lang3/builder
```

Если мы примем во внимание, что ресурс пакета из файла JAR начинается с префикса `jar:`, то мы можем написать метод, который будет проводить различие между ними следующим образом:

```
private static final String JAR_PREFIX = "jar:";

public static List<Class<?>> fetchClassesFromPackage(
    String packageName) throws URISyntaxException, IOException {

    List<Class<?>> classes = new ArrayList<>();
    String packagePath = packageName.replace('.', '/');

    URL resource = ClassLoader
        .getSystemClassLoader().getResource(packagePath);
```

```
if (resource != null) {
    if (resource.toString().startsWith(JAR_PREFIX)) {
        classes.addAll(fetchClassesFromJar(resource, packageName));
    } else {
        File file = new File(resource.toURI());
        classes.addAll(fetchClassesFromDirectory(file, packageName));
    }
} else {
    throw new RuntimeException("Ресурс не найден для пакета: "
        + packageName);
}

return classes;
}
```

Итак, если заданный пакет находится в файле JAR, то мы вызываем еще один вспомогательный метод, `fetchClassesFromJar()`; в противном случае мы вызываем вспомогательный метод `fetchClassesFromDirectory()`. Как следует из их названий, эти помощники знают, как получать классы заданного пакета из файла JAR или из каталога.

В общих чертах, эти два метода представляют собой лишь фрагменты *спагетти-кода*, предназначенные для выявления файлов, которые имеют расширение `.class`. Каждый класс пропускается через метод `Class.forName()` с целью обеспечения, чтобы класс возвращался как тип `Class`, а не как тип `String`. Оба метода имеются в исходном коде, прилагаемом к этой книге.

Как насчет перечисления классов из пакетов, которые отсутствуют в загрузчике системных классов, например пакета из внешнего файла JAR? Удобный способ для выполнения этой задачи опирается на класс `URLClassLoader`. Этот класс используется для загрузки классов и ресурсов из поискового пути URL-адресов, которые ссылаются как на файлы JAR, так и на каталоги. Мы будем иметь дело только с файлами JAR, но это довольно просто сделать и для каталогов.

Таким образом, исходя из заданного пути, нам нужно получить все файлы JAR и вернуть их в виде массива `URL[]` (этот массив необходим для определения класса `URLClassLoader`). Например, мы можем опереться на метод `Files.find()`, чтобы пройти по заданному пути и получить все файлы JAR, как вот здесь:

```
public static URL[] fetchJarsUrlsFromClasspath(Path classpath)
    throws IOException {

    List<URL> urlsOfJars = new ArrayList<>();
    List<File> jarFiles = Files.find(
        classpath,
        Integer.MAX_VALUE,
        (path, attr) -> !attr.isDirectory() &&
```



```
        path.toString().toLowerCase().endsWith(JAR_EXTENSION))
        .map(Path::toFile)
        .collect(Collectors.toList());

for (File jarFile: jarFiles) {
    try {
        urlsOfJars.add(jarFile.toURI().toURL());
    } catch (MalformedURLException e) {
        logger.log(Level.SEVERE, "Плохой UR-адрес для {0} {1}",
            new Object[] {
                jarFile, e
            });
    }
}

return urlsOfJars.toArray(URL[]::new);
}
```

Обратите внимание, что мы сканируем все подкаталоги, начиная с заданного пути. Конечно, это решение является проектным, и легко можно параметризовать глубину поиска. Теперь давайте выберем файлы JAR из папки `tomcat8/lib` (для этого не нужно специально устанавливать Tomcat; просто используйте любой другой локальный каталог файлов JAR и внесите соответствующие изменения):

```
URL[] urls = Packages.fetchJarsUrlsFromClasspath(
    Path.of("D:/tomcat8/lib"));
```

Теперь мы можем создать экземпляр класса `URLClassLoader`:

```
URLClassLoader urlClassLoader = new URLClassLoader(
    urls, Thread.currentThread().getContextClassLoader());
```

В результате будет создан новый объект `URLClassLoader` для заданных URL-адресов, и текущий загрузчик классов будет использоваться для делегирования (второй аргумент также может быть `null`). Наш массив `URL[]` указывает только на файлы JAR, но в качестве общего правила, считается, что любой URL-адрес с префиксом `jar:` ссылается на файл JAR, а любой URL-адрес с префиксом `file:`, который заканчивается на `/`, ссылается на каталог.

Один из файлов JAR, который присутствует в папке `tomcat8/lib`, называется `tomcatjdbc.jar`. В этом файле JAR находится пакет под названием `org.apache.tomcat.jdbc.pool`. Давайте перечислим классы этого пакета:

```
List<Class?>> classes = Packages.fetchClassesFromPackage(
    "org.apache.tomcat.jdbc.pool", urlClassLoader);
```

Метод `fetchClassesFromPackage()` является помощником, который просто сканирует массив `URL[]` объектов класса `URLClassLoader` и получает классы, находящиеся в заданном пакете. Его код имеется в исходном коде, прилагаемом к этой книге.

Инспектирование пакетов внутри модулей

Если воспользоваться модульным принципом Java 9, то наши пакеты будут располагаться внутри модулей. Например, если у нас есть класс с именем `Manager` в пакете `com.management` в модуле под названием `org.tournament`, то мы можем получить все пакеты этого модуля следующим образом:

```
Manager mgt = new Manager();  
Set<String> packages = mgt.getClass().getModule().getPackages();
```

В дополнение к этому, если мы хотим создать класс, то нам нужна следующая разновидность метода `Class.forName()`:

```
Class<?> clazz = Class.forName(mgt.getClass()  
.getModule(), "com.management.Manager");
```

Имейте в виду, что каждый модуль представлен на диске как каталог с тем же именем. Например, модуль `org.tournament` на диске представляет собой папку с таким названием. Кроме того, каждый модуль отображается как отдельный файл JAR с этим именем (например, `org.tournament.jar`). Зная эти координаты, довольно просто адаптировать код из этого раздела так, чтобы он перечислял все классы заданного пакета конкретного модуля.

150. Инспектирование классов

Используя API рефлексии Java, мы можем осмотреть детали класса — имя класса объекта, модификаторы, конструкторы, методы, поля, имплементированные интерфейсы и т. д.

Допустим, что мы имеем следующий класс `Pair`:

```
public final class Pair<L, R> extends Tuple implements Comparable {  
    final L left;  
    final R right;  
  
    public Pair(L left, R right) {  
        this.left = left;  
        this.right = right;  
    }  
  
    public class Entry<L, R> {}  
    ...  
}
```

Давайте также допустим, что у нас есть его экземпляр:

```
Pair pair = new Pair(1, 1);
```

Теперь воспользуемся рефлексией, чтобы получить имя класса `Pair`.

Получение имени класса *Pair* посредством экземпляра

Имея экземпляр (объект) класса *Pair*, мы можем узнать имя его класса, вызвав метод `getClass()`, а также методы `Class.getName()`, `getSimpleName()` и `getCanonicalName()`, как показано в следующем примере:

```
Class<?> clazz = pair.getClass();

// modern.challenge.Pair
System.out.println("Имя: " + clazz.getName());

// Pair
System.out.println("Простое имя: " + clazz.getSimpleName());

// modern.challenge.Pair
System.out.println("Каноническое имя: " + clazz.getCanonicalName());
```



Анонимный класс не имеет простых и канонических имен.

Обратите внимание, что метод `getSimpleName()` возвращает невалифицированное имя класса. В качестве альтернативы мы можем получить класс следующим образом:

```
Class<Pair> clazz = Pair.class;
Class<?> clazz = Class.forName("modern.challenge.Pair");
```

Получение модификаторов класса *Pair*

Для того чтобы получить модификаторы (`public`, `protected`, `private`, `final`, `static`, `abstract` и `interface`) класса, мы можем вызвать метод `Class.getModifiers()`. Этот метод возвращает значение типа `int`, представляющее каждый модификатор в виде флагового бита. Для декодирования результата мы опираемся на класс `Modifier`, как показано ниже:

```
int modifiers = clazz.getModifiers();

System.out.println("Is public? " + Modifier.isPublic(modifiers)); // true
System.out.println("Is final? " + Modifier.isFinal(modifiers)); // true
System.out.println("Is abstract? " + Modifier.isAbstract(modifiers)); // false
```

Получение имплементированных интерфейсов класса *Pair*

Для того чтобы получить интерфейсы, непосредственно имплементируемые классом, либо интерфейс, представленный объектом, мы просто вызываем метод `Class.getInterfaces()`. Этот метод возвращает массив. Поскольку класс *Pair* импле-

ментирует один интерфейс (`Comparable`), возвращаемый массив будет содержать один-единственный элемент:

```
Class<?>[] interfaces = clazz.getInterfaces();

// interface java.lang.Comparable
System.out.println("Интерфейсы: " + Arrays.toString(interfaces));

// Comparable
System.out.println("Простое имя интерфейса: "
    + interfaces[0].getSimpleName());
```

Получение конструкторов класса *Pair*

Публичные конструкторы класса могут быть получены посредством метода `Class.getConstructors()`. Возвращаемым результатом является `Constructor<?>[]`:

```
Constructor<?>[] constructors = clazz.getConstructors();

// public modern.challenge.Pair(java.lang.Object,java.lang.Object)
System.out.println("Constructors: " + Arrays.toString(constructors));
```



Для получения всех объявленных конструкторов (например, приватных и защищенных конструкторов) нужно вызывать метод `getDeclaredConstructors()`. При поиске некоторого конструктора следует вызывать методы `getConstructor(Class<?>... parameterTypes)` или `getDeclaredConstructor(Class<?>... parameterTypes)`.

Получение полей класса *Pair*

Все поля класса доступны посредством метода `Class.getDeclaredFields()`, который возвращает массив объектов `Field`:

```
Field[] fields = clazz.getDeclaredFields();

// final java.lang.Object modern.challenge.Pair.left
// final java.lang.Object modern.challenge.Pair.right
System.out.println("Fields: " + Arrays.toString(fields));
```

Для получения фактического имени полей мы можем легко предоставить вспомогательный метод:

```
public static List<String> getFieldNames(Field[] fields) {
    return Arrays.stream(fields)
        .map(Field::getName)
        .collect(Collectors.toList());
}
```

Теперь мы получаем только имена полей:

```
List<String> fieldsName = getFieldNames(fields);

// left, right
System.out.println("Имена полей: " + fieldsName);
```

Получить значение поля можно посредством общего метода `Object get(Object obj)` и с помощью связки методов `getFoo()` (подробнее см. документацию). `obj` представляет статическое или экземплярное поле. Например, допустим, что класс `ProcedureOutputs` имеет приватное поле с именем `callableStatement` и типом `CallableStatement`. Воспользуемся методом `Field.get()`, чтобы обратиться к этому полю для проверки, является ли `CallableStatement` закрытым (`isClosed()`):

```
ProcedureOutputs procedureOutputs
    = storedProcedure.unwrap(ProcedureOutputs.class);

Field csField = procedureOutputs.getClass()
    .getDeclaredField("callableStatement");
csField.setAccessible(true);

CallableStatement cs
    = (CallableStatement) csField.get(procedureOutputs);

System.out.println("Закрытое? " + cs.isClosed());
```



Для получения только публичных полей следует вызывать метод `getFields()`. Для поиска некоторого поля следует вызывать методы `getField(String fieldName)` или `getDeclaredField(String name)`.

Получение методов класса *Pair*

Публичные методы класса доступны посредством метода `Class.getMethods()`. Этот метод возвращает массив объектов `Method`:

```
Method[] methods = clazz.getMethods();
// public boolean modern.challenge.Pair.equals(java.lang.Object)
// public int modern.challenge.Pair.hashCode()
// public int modern.challenge.Pair.compareTo(java.lang.Object)
// ...

System.out.println("Методы: " + Arrays.toString(methods));
```

Для получения фактического имени методов мы можем быстро предоставить вспомогательный метод:

```
public static List<String> getMethodNames(Method[] methods) {
    return Arrays.stream(methods)
        .map(Method::getName)
        .collect(Collectors.toList());
}
```

Теперь мы получаем только имена методов:

```
List<String> methodName = getMethodNames(methods);

// equals, hashCode, compareTo, wait, wait,
// wait, toString, getClass, notify, notifyAll
System.out.println("Имена методов: " + methodName);
```



Для получения всех объявленных методов (например, приватных и защищенных), нужно вызывать метод `getDeclaredMethods()`. Для поиска некоторого метода следует вызывать методы `getMethod(String name, Class<?>... parameterTypes)` или `getDeclaredMethod(String name, Class<?>... parameterTypes)`.

Получение модуля класса *Pair*

Если мы воспользуемся модульным принципом JDK 9, то наши классы будут располагаться внутри модулей. Класс *Pair* отсутствует в модуле, но мы можем легко получить модуль класса посредством метода JDK 9 `Class.getModule()` (если класса нет в модуле, то этот метод возвращает `null`):

```
// null, поскольку Pair не находится в модуле
Module module = clazz.getModule();
```

Получение надкласса класса *Pair*

Класс *Pair* расширяет класс *Tuple*; следовательно, класс *Tuple* является надклассом класса *Pair*. Мы можем получить его посредством метода `Class.getSuperclass()` так:

```
Class<?> superClass = clazz.getSuperclass();
// modern.challenge.Tuple
System.out.println("Надкласс: " + superClass.getName());
```

Получение имени некоторого типа

Начиная с JDK 8, мы можем получить информативное строковое значение с именем определенного типа. Методы `getName()`, `getSimpleName()` или `getCanonicalName()` возвращают указанное строковое значение.

◆ Для примитивов все три метода возвращают следующее:

```
System.out.println("Тип: " + int.class.getTypeName()); // int
```

◆ Для класса *Pair* методы `getName()` и `getCanonicalName()` возвращают следующее:

```
// modern.challenge.Pair
System.out.println("Имя типа: " + clazz.getTypeName());
```

◆ Для внутренних классов (например, таких как *Entry* для класса *Pair*) метод `getName()` возвращает следующее:

```
// modern.challenge.Pair$Entry
System.out.println("Имя типа: " + Pair.Entry.class.getTypeName());
```

◆ Для анонимного класса метод `getName()` возвращает следующее:

```
Thread thread = new Thread() {
    public void run() {
        System.out.println("Дочерняя нить исполнения");
    }
};
```

```
// modern.challenge.Main$1
System.out.println("Имя типа анонимного класса: "
    + thread.getClass().getTypeName());
```

◆ Для массивов метод `getCanonicalName()` возвращает следующее:

```
Pair[] pairs = new Pair[10];
// modern.challenge.Pair[]
System.out.println("Имя типа массива: " + pairs.getClass().getTypeName());
```

Получение строк, описывающих класс

Начиная с JDK 8, мы можем получить краткое описание класса (содержащего модификаторы, имя, параметры типов и т. д.) посредством метода `Class.toGenericString()`.

Давайте взглянем на несколько примеров:

```
// public final class modern.challenge.Pair<L,R>
System.out.println("Описание класса Pair: " + clazz.toGenericString());

// public abstract interface java.lang Runnable
System.out.println("Описание интерфейса Runnable: "
    + Runnable.class.toGenericString());

// public abstract interface java.util.Map<K,V>
System.out.println("Описание интерфейса Map: " + Map.class.toGenericString());
```

Получение строки с дескриптором типа для класса

Начиная с JDK 12, мы можем получить дескриптор типа класса как объект `String` посредством метода `Class.descriptorString()`:

```
// Lmodern/challenge/Pair;
System.out.println("Дескриптор типа для класса Pair: "
    + clazz.descriptorString());

// Ljava/lang/String;
System.out.println("Дескриптор типа для класса String: "
    + String.class.descriptorString());
```

Получение типа компонента для массива

Только для массивов JDK 12 предусматривает метод `Class<?> componentType()`. Он возвращает тип компонента для массива, как показано в следующих двух примерах:

```
Pair[] pairs = new Pair[10];
String[] strings = new String[] {"1", "2", "3"};

// class modern.challenge.Pair
System.out.println("Тип компонента для массива объектов Pair[]: "
    + pairs.getClass().componentType());
```

```
// class java.lang.String
System.out.println("Тип компонента для массива объектов String[]: "
    + strings.getClass().componentType());
```

Получение класса для индексируемого типа, тип компонента которого описывается классом *Pair*

Начиная с JDK 12, мы можем получать класс для индексируемого типа, тип компонента которого описывается заданным классом, посредством метода `Class.arrayType()`:

```
Class<?> arrayClazz = clazz.arrayType();
```

```
// modern.challenge.Pair<L,R>[]
System.out.println("Тип массива: " + arrayClazz.toGenericString());
```

151. Создание экземпляра класса посредством отрефлексированного конструктора

Мы можем создавать экземпляры класса посредством метода `Constructor.newInstance()` с использованием API рефлексии Java.

Рассмотрим класс, который имеет четыре конструктора:

```
public class Car {
    private int id;
    private String name;
    private Color color;

    public Car() {}

    public Car(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public Car(int id, Color color) {
        this.id = id;
        this.color = color;
    }

    public Car(int id, String name, Color color) {
        this.id = id;
        this.name = name;
        this.color = color;
    }

    // геттеры и сеттеры опущены для краткости
}
```


Экземпляр класса `Car` может быть создан посредством одного из этих четырех конструкторов. Класс `Constructor` выставляет наружу метод, который берет типы параметров конструктора и возвращает объект `Constructor`, отражающий совпадающий конструктор. Этот метод вызывается `getConstructor(Class<?>... parameterTypes)`.

Давайте вызовем каждый из приведенных выше конструкторов:

```
Class<Car> clazz = Car.class;

Constructor<Car> emptyCnstr = clazz.getConstructor();

Constructor<Car> idNameCnstr = clazz.getConstructor(int.class, String.class);

Constructor<Car> idColorCnstr = clazz.getConstructor(int.class, Color.class);

Constructor<Car> idNameColorCnstr
    = clazz.getConstructor(int.class, String.class, Color.class);
```

Далее метод `Constructor.newInstance(Object... initargs)` может вернуть экземпляр класса `Car`, который соответствует вызванному конструктору:

```
Car carViaEmptyCnstr = emptyCnstr.newInstance();

Car carViaIdNameCnstr = idNameCnstr.newInstance(1, "Dacia");

Car carViaIdColorCnstr = idColorCnstr.newInstance(1, new Color(0, 0, 0));

Car carViaIdNameColorCnstr = idNameColorCnstr
    .newInstance(1, "Dacia", new Color(0, 0, 0));
```

Теперь самое время посмотреть, как создавать экземпляр приватного конструктора посредством рефлексии.

Создание экземпляра класса посредством приватного конструктора

API рефлексии Java может также использоваться для создания экземпляра класса посредством приватного конструктора. Например, предположим, что у нас есть служебный класс `Cars`. Следуя рекомендациям, мы определим этот класс как финальный и с приватным конструктором, чтобы не допускать экземпляров:

```
public final class Cars {
    private Cars() {}
    // статические члены
}
```

Получить этот конструктор можно посредством метода `Class.getDeclaredConstructor()` следующим образом:

```
Class<Cars> carsClass = Cars.class;
Constructor<Cars> emptyCarsCnstr = carsClass.getDeclaredConstructor();
```

Вызов `newInstance()` в этом экземпляре будет выбрасывать исключение `IllegalAccessException`, т. к. вызываемый конструктор имеет приватный доступ. Однако рефлексия Java позволяет нам модифицировать уровень доступа посредством флагового метода `Constructor.setAccessible()`. На этот раз создание экземпляра работает, как и ожидалось:

```
emptyCarsCnstr.setAccessible(true);
Cars carsViaEmptyCnstr = emptyCarsCnstr.newInstance();
```

Для того чтобы заблокировать этот подход, рекомендуется выбрасывать ошибку из приватного конструктора, как показано ниже:

```
public final class Cars {
    private Cars() {
        throw new AssertionError("Не получается создать экземпляр");
    }

    // статические члены
}
```

На этот раз попытка создания экземпляра завершится ошибкой `AssertionError`.

Создание экземпляра класса из файла JAR

Предположим, что в папке `D:/Java ModernChallenge/Code/lib/` у нас есть файл JAR библиотеки Guava, и мы хотим создать экземпляр `CountingInputStream` и прочитать 1 байт из этого файла.

Сначала мы определяем массив `URL[]` для файла JAR библиотеки Guava следующим образом:

```
URL[] classLoaderUrls = new URL[] {
    new URL("file:///D:/Java Modern Challenge/Code/lib/guava-16.0.1.jar")
};
```

Затем мы определяем `URLClassLoader` для этого массива `URL[]`:

```
URLClassLoader urlClassLoader = new URLClassLoader(classLoaderUrls);
```

Далее мы загружаем целевой класс (`CountingInputStream` — это класс, который подсчитывает число байтов, считываемых из потока `InputStream`):

```
Class<?> cisClass = urlClassLoader.loadClass(
    "com.google.common.io.CountingInputStream");
```

После загрузки целевого класса мы можем получить его конструктор (`CountingInputStream` имеет один конструктор, который обортывает заданный поток `InputStream`):

```
Constructor<?> constructor = cisClass.getConstructor(InputStream.class);
```

Далее мы можем создать экземпляр класса `CountingInputStream` посредством этого конструктора:

```
Object instance = constructor.newInstance(
    new FileInputStream(Path.of("test.txt").toFile()));
```

С целью обеспечения операционности возвращаемого экземпляра, давайте вызовем два его метода (метод `read()` сразу читает 1 байт, тогда как метод `getCount()` возвращает число прочитанных байтов):

```
Method readMethod = cisClass.getMethod("read");
```

```
Method countMethod = cisClass.getMethod("getCount");
```

Далее прочитаем 1 байт и посмотрим, что возвращает `getCount()`:

```
readMethod.invoke(instance);
```

```
Object readBytes = countMethod.invoke(instance);
```

```
System.out.println("Прочитано байтов (должно быть 1): " + readBytes); // 1
```

Полезные фрагменты кода

В качестве бонуса давайте рассмотрим несколько фрагментов кода, которые обычно необходимы при работе с рефлексией и конструкторами.

Сначала давайте получим число имеющихся конструкторов:

```
Class<Car> clazz = Car.class;
```

```
Constructor<?>[] cnstrs = clazz.getConstructors();
```

```
System.out.println("Класс Car имеет " + cnstrs.length + " конструкторов"); // 4
```

Теперь посмотрим, сколько параметров имеет каждый из этих четырех конструкторов:

```
for (Constructor<?> cnstr : cnstrs) {
    int paramCount = cnstr.getParameterCount();
    System.out.println("\nКонструктор с " + paramCount + " параметрами");
}
```

Для того чтобы получить подробную информацию о каждом параметре конструктора, мы можем вызвать метод `Constructor.getParameters()`. Этот метод возвращает массив объектов `Parameter` (этот класс был добавлен в JDK 8 и предоставляет полный список методов для разложения параметра):

```
for (Constructor<?> cnstr : cnstrs) {
    Parameter[] params = cnstr.getParameters();
    ...
}
```

Если нам нужно просто узнать типы параметров, то метод

`Constructor.getParameterTypes()` сделает эту работу:

```
for (Constructor<?> cnstr : cnstrs) {
    Class<?>[] typesOfParams = cnstr.getParameterTypes();
    ...
}
```

152. Извлечение аннотации типа получателя

Начиная с JDK 8, мы можем использовать явные параметры *получателя*. В общих чертах это означает, что мы можем объявлять экземплярный метод, который берет параметр обрамляющего типа с ключевым словом `this` языка Java.

Посредством явных параметров *получателя* мы можем прикреплять аннотации типов к `this`. Допустим, что у нас есть следующая аннотация:

```
@Target({ElementType.TYPE_USE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Ripe {}
```

Давайте применим ее для аннотирования ключевого слова `this` в методе `eat()` класса `Melon` (дыня):

```
public class Melon {
    ...
    public void eat(@Ripe Melon this) {}
    ...
}
```

Другими словами, мы можем вызвать метод `eat()` только в том случае, если экземпляр класса `Melon` представляет собой спелую дыню:

```
Melon melon = new Melon("Gac", 2000);
```

```
// работает, только если дыня является спелой
melon.eat();
```

Получить аннотацию на явном параметре получателя с помощью рефлексии можно посредством метода JDK 8 `java.lang.reflect.Executable.getAnnotatedReceiverType()`. Этот метод также доступен в классах `Constructor` и `Method`, поэтому мы можем использовать его следующим образом:

```
Class<Melon> clazz = Melon.class;
Method eatMethod = clazz.getDeclaredMethod("eat");
```

```
AnnotatedType annotatedType = eatMethod.getAnnotatedReceiverType();
```

```
// modern.challenge.Melon
System.out.println("Тип: " + annotatedType.getType().getTypeName());
```

```
// [@modern.challenge.Ripe()]
System.out.println("Аннотации: "
    + Arrays.toString(annotatedType.getAnnotations()));
```

```
// [interface java.lang.reflect.AnnotatedType]
System.out.println("Класс, имплементирующий интерфейсы: "
    + Arrays.toString(annotatedType.getClass().getInterfaces()));
```

```
AnnotatedType annotatedOwnerType = annotatedType.getAnnotatedOwnerType();
```

```
// null
System.out.println("\nАннотированный тип владельца: " + annotatedOwnerType);
```

153. Получение синтетических и мостовых конструкций

Используя синтетические конструкции, мы можем понять практически любую конструкцию, добавляемую компилятором. Точнее, в соответствии со спецификацией языка Java: *любые конструкции, вводимые компилятором Java, которые не имеют соответствующей конструкции в исходном коде, должны быть помечены как синтетические, за исключением конструкторов по умолчанию, метода инициализации класса и значений, а также методов `valueOf()` класса `Enum`*.

Существуют разные виды *синтетических конструкций* (например, поля, методы и конструкторы), но давайте взглянем на пример *синтетического поля*. Допустим, что мы имеем следующий класс:

```
public class Melon {
    ...
    public class Slice {}
    ...
}
```

Обратите внимание, что у нас есть внутренний класс под названием `Slice`. Когда код компилируется, компилятор изменяет этот класс, добавляя *синтетическое поле*, которое должно ссылаться на класс верхнего уровня. Это синтетическое поле обеспечивает доступ к членам обрамляющего класса из вложенного класса.

Для того чтобы проверить наличие этого синтетического поля, давайте извлечем все объявленные поля и подсчитаем их:

```
Class<Melon.Slice> clazzSlice = Melon.Slice.class;
Field[] fields = clazzSlice.getDeclaredFields();

// 1
System.out.println("Число полей: " + fields.length);

Даже если мы явно не объявляли какие-либо поля, обратите внимание, что все-таки
об одном поле было сообщено. Давайте посмотрим, является ли оно синтетиче-
ским, и взглянем на его название:

// true
System.out.println("Является синтетическим: " + fields[0].isSynthetic());

// this$0
System.out.println("Имя: " + fields[0].getName());
```



Аналогично этому примеру мы можем проверить, является ли метод или конструктор синтетическим, посредством методов `Method.isSynthetic()` и `Constructor.isSynthetic()`.

Теперь поговорим о *мостовых методах*. Эти методы также являются синтетическими, и их цель — обработка *стирания типов* у обобщений (джереников, generics).

Рассмотрим следующий класс `Melon`:

```
public class Melon implements Comparator<Melon> {
    @Override
    public int compare(Melon m1, Melon m2) {
        return Integer.compare(m1.getWeight(), m2.getWeight());
    }
    ...
}
```

Здесь мы имплементируем интерфейс `Comparator` и переопределяем его метод `compare()`. Кроме того, мы явно указали, что метод `compare()` берет два экземпляра класса `Melon`. Компилятор продолжит выполнять *стирание типов* и создаст новый метод, который берет два объекта, как показано ниже:

```
public int compare(Object m1, Object m2) {
    return compare((Melon) m1, (Melon) m2);
}
```

Этот метод именуется *синтетическим мостовым* методом. Мы его не можем видеть, а вот API рефлексии Java может:

```
Class<Melon> clazz = Melon.class;
Method[] methods = clazz.getDeclaredMethods();
Method compareBridge = Arrays.asList(methods).stream()
    .filter(m -> m.isSynthetic() && m.isBridge())
    .findFirst()
    .orElseThrow();
```

```
// public int modern.challenge.Melon.compare(
// java.lang.Object, java.lang.Object)
System.out.println(compareBridge);
```

154. Проверка переменного числа аргументов

В языке Java метод может получать переменное число аргументов, если его сигнатура содержит аргумент типа `varargs`.

Скажем, метод `plantation()` берет переменное число аргументов, например `Seed...seeds`:

```
public class Melon {
    ...
    public void plantation(String type, Seed...seeds) {}
    ...
}
```

Так вот, API рефлексии Java может распознать, поддерживает ли этот метод переменное число аргументов, посредством метода `Method.isVarArgs()` следующим образом:

```
Class<Melon> clazz = Melon.class;
Method[] methods = clazz.getDeclaredMethods();
```

```
for (Method method: methods) {  
    System.out.println("Имя метода: " + method.getName()  
        + " varargs? " + method.isVarArgs());  
}
```

Вы получите результат, подобный следующему:

```
Имя метода: plantation, varargs? true  
Имя метода: getWeight, varargs? false  
Имя метода: toString, varargs? false  
Имя метода: getType, varargs? false
```

155. Проверка методов по умолчанию

Среда Java 8 обогатила концепцию интерфейсов методами по умолчанию. Эти методы написаны внутри интерфейсов и имеют имплементацию по умолчанию. Например, интерфейс `Slicer` имеет метод по умолчанию под названием `slice()`:

```
public interface Slicer {  
    public void type();  
  
    default void slice() {  
        System.out.println("slice");  
    }  
}
```

Теперь любая имплементация интерфейса `Slicer` должна имплементировать метод `type()` и опционально может переопределять метод `slice()` либо опираться на имплементацию по умолчанию.

API рефлексии Java может идентифицировать метод по умолчанию посредством флагового метода `Method.isDefault()`:

```
Class<Slicer> clazz = Slicer.class;  
Method[] methods = clazz.getDeclaredMethods();
```

```
for (Method method: methods) {  
    System.out.println("Имя метода: " + method.getName()  
        + ", является методом по умолчанию? " + method.isDefault());  
}
```

Мы получим следующий результат:

```
Имя метода: type, является методом по умолчанию? false  
Имя метода: slice, является методом по умолчанию? true
```

156. Управление гнездовым доступом посредством рефлексии

Среди характерных средств JDK 11 имеется несколько *горячих точек* (изменений на уровне байт-кода). Одна из таких *горячих точек* известна как JEP 181, или **управление гнездовым доступом** (nest-based access control), или **гнезда** (nests).

В сущности, термин «гнездо» (nest) определяет новый контекст управления доступом, который *позволяет классам, которые логически являются частью одной и той же сущности кода, но компилируются с отдельными файлами классов, обращаться к приватным членам друг друга без необходимости в том, чтобы компиляторы вставляли расширяющие доступность мостовые методы.*

Другими словами, *гнезда* позволяют компилировать вложенные классы в разные файлы классов, принадлежащие одному и тому же обрамляющему классу. Затем они получают разрешение обращаться к приватным классам друг друга без использования *синтетических/мостовых* методов.

Рассмотрим следующий код:

```
public class Car {
    private String type = "Dacia";

    public class Engine {
        private String power = "80 hp";

        public void addEngine() {
            System.out.println("Добавить двигатель " + power
                + « в автомашину типа « + type);
        }
    }
}
```

Давайте выполним `javap` (дизассемблерный инструмент файлов классов Java, который позволяет нам анализировать байт-код) для `Car.class` в JDK 10. На рис. 7.1 показан снимок экрана с важной частью этого кода.



```
JDK 10                                     javap
Compiled from "Car.java"
public class modern.challenge.Car {
    public modern.challenge.Car();
    public static modern.challenge.Car newCar(java.lang.String, java.lang.String)
        throws java.lang.NoSuchFieldException, java.lang.IllegalArgumentException, java.
        lang.IllegalAccessException;
    static java.lang.String access$000(modern.challenge.Car);
}
```

Рис. 7.1

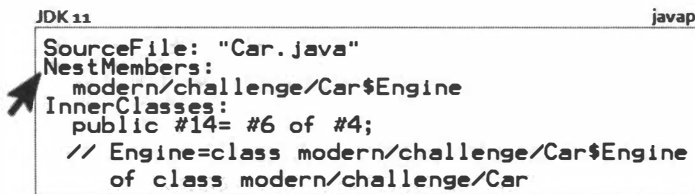
Как мы видим, чтобы обратиться к полю `Car.type` обрамляющего класса из метода `Engine.addEngine()`, Java изменил код и добавил *мостовой* пакетно-приватный (package-private) метод под названием `access$000()`. В общих чертах, он генерируется синтетически, и его можно увидеть посредством рефлексии, используя методы `Method.isSynthetic()` и `Method.isBridge()`.

Даже если мы видим (или воспринимаем) классы `Car` (внешний класс) и `Engine` (вложенный класс) как находящиеся в одном и том же классе, они компилируются в разные файлы (`Car.class` и `Car$Engine.class`). Значит, стоит ожидать, что внешние и вложенные классы могут обращаться к приватным членам друг друга.

Но это невозможно, если классы находятся в отдельных файлах. Для того чтобы оправдать наши ожидания, Java добавляет *синтетический мостовой* пакетно-приватный метод `access$000()`.

Однако Java 11 вводит *гнездовой* контекст управления доступом, который обеспечивает поддержку приватного доступа внутри внешних и вложенных классов. На этот раз внешние и вложенные классы связаны с двумя атрибутами и образуют гнездо (мы говорим, что они являются *соседями по гнезду*). В общих чертах вложенные классы связаны с атрибутом `NestMembers`, тогда как внешний класс связан с атрибутом `NestHost`. Никакой дополнительный *синтетический* метод не генерируется.

На рис. 7.2 видно, как `javap` выполняется в JDK 11 для `Car.class` (обратите внимание на атрибут `NestMembers`).

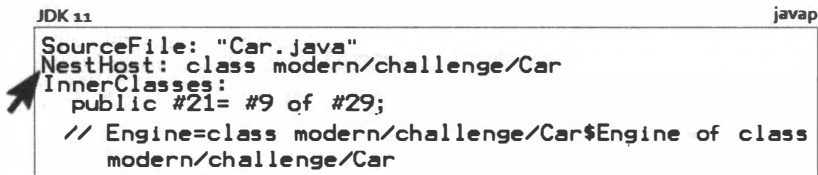


```

JDK 11                                     javap
SourceFile: "Car.java"
NestMembers:
  modern/challenge/Car$Engine
InnerClasses:
  public #14= #6 of #4;
  // Engine=class modern/challenge/Car$Engine
    of class modern/challenge/Car
  
```

Рис. 7.2

На рис. 7.3 показаны данные `javap` в JDK 11 для `Car$Engine.class` (обратите внимание на атрибут `NestHost`).



```

JDK 11                                     javap
SourceFile: "Car.java"
NestHost: class modern/challenge/Car
InnerClasses:
  public #21= #9 of #29;
  // Engine=class modern/challenge/Car$Engine of class
    modern/challenge/Car
  
```

Рис. 7.3

Доступ посредством API рефлексии

Без управления гнездовым доступом возможности рефлексии также ограничены. Например, до JDK 11 следующий ниже фрагмент кода будет выбрасывать исключение `IllegalAccessException`:

```

Car newCar = new Car();
Engine engine = newCar.new Engine();

Field powerField = Engine.class.getDeclaredField("power");
powerField.set(engine, power);
  
```

Мы можем позволить доступ путем явного вызова метода

```
powerField.setAccessible(true);  
  
...  
Field powerField = Engine.class.getDeclaredField("power");  
powerField.setAccessible(true);  
powerField.set(engine, power);  
  
...
```

Начиная с JDK 11, вызывать метод `setAccessible()` не требуется.

Кроме того, JDK 11 идет в комплекте с тремя методами, которые обогащают API рефлексии Java поддержкой *гнезд*. Этими методами являются `Class.getNestHost()`, `Class.getNestMembers()` и `Class.isNestmateOf()`.

Возьмем класс `Melon` с несколькими вложенными классами (`Slice`, `Peeler` и `Juicer`):

```
public class Melon {  
    ...  
    public class Slice {  
        public class Peeler {}  
    }  
  
    public class Juicer {}  
    ...  
}
```

Теперь определим `Class` для каждого из них:

```
Class<Melon> clazzMelon = Melon.class;  
Class<Melon.Slice> clazzSlice = Melon.Slice.class;  
Class<Melon.Juicer> clazzJuicer = Melon.Juicer.class;  
Class<Melon.Slice.Peeler> clazzPeeler = Melon.Slice.Peeler.class;
```

Для того чтобы увидеть `NestHost` каждого класса, нам нужно вызвать метод

```
Class.getNestHost():  
  
// class modern.challenge.Melon  
Class<?> nestClazzOfMelon = clazzMelon.getNestHost();  
  
// class modern.challenge.Melon  
Class<?> nestClazzOfSlice = clazzSlice.getNestHost();  
  
// class modern.challenge.Melon  
Class<?> nestClazzOfPeeler = clazzPeeler.getNestHost();  
  
// class modern.challenge.Melon  
Class<?> nestClazzOfJuicer = clazzJuicer.getNestHost();
```

Здесь следует подчеркнуть два момента. Во-первых, обратите внимание, что гнездовым классом `NestHost` класса `Melon` является сам класс `Melon`. Во-вторых, гнездовым классом `NestHost` класса `Peeler` является класс `Melon`, а не `Peeler`. Поскольку

класс `Peeler` является внутренним, мы можем подумать, что его `NestHost` является класс `Peeler`, но это предположение неверно.

Теперь давайте перечислим членов `NestMembers` каждого класса:

```
Class<?>[] nestMembersOfMelon = clazzMelon.getNestMembers();
Class<?>[] nestMembersOfSlice = clazzSlice.getNestMembers();
Class<?>[] nestMembersOfJuicer = clazzJuicer.getNestMembers();
Class<?>[] nestMembersOfPeeler = clazzPeeler.getNestMembers();
```

Все они вернут тех же самых членов гнезда `NestMembers`:

```
[class modern.challenge.Melon, class modern.challenge.Melon$Juicer,
 class modern.challenge.Melon$Slice, class
 modern.challenge.Melon$Slice$Peeler]
```

Наконец, давайте проверим *соседей по гнезду*:

```
boolean melonIsNestmateOfSlice = clazzMelon.isNestmateOf(clazzSlice); // true
boolean melonIsNestmateOfJuicer = clazzMelon.isNestmateOf(clazzJuicer); // true
boolean melonIsNestmateOfPeeler = clazzMelon.isNestmateOf(clazzPeeler); // true
boolean sliceIsNestmateOfJuicer = clazzSlice.isNestmateOf(clazzJuicer); // true
boolean sliceIsNestmateOfPeeler = clazzSlice.isNestmateOf(clazzPeeler); // true
boolean juicerIsNestmateOfPeeler =
    clazzJuicer.isNestmateOf(clazzPeeler); // true
```

157. Рефлексия для геттеров и сеттеров

Напомним, что **геттеры** и **сеттеры** — это методы (так называемые аксессоры), которые используются для доступа к полям класса (например, к приватным полям).

Сначала давайте посмотрим, как получить существующие геттеры и сеттеры. Позже мы попытаемся сгенерировать недостающие геттеры и сеттеры с помощью рефлексии.

Получение геттеров и сеттеров

Существует несколько вариантов решения задачи получения геттеров и сеттеров класса посредством рефлексии. Допустим, что мы хотим получить геттеры и сеттеры класса `Melon`:

```
public class Melon {
    private String type;
    private int weight;
    private boolean ripe;
    ...

    public String getType() {
        return type;
    }
}
```

```
public void setType(String type) {
    this.type = type;
}

public int getWeight() {
    return weight;
}

public void setWeight(int weight) {
    this.weight = weight;
}

public boolean isRipe() {
    return ripe;
}

public void setRipe(boolean ripe) {
    this.ripe = ripe;
}
...
}
```

Начнем с варианта решения, который получает все объявленные методы класса посредством рефлексии (например, с помощью метода `Class.getDeclaredMethods()`). Теперь обойдем массив `Method[]` в цикле и отфильтруем его в соответствии с ограничениями, которые являются специфичными для геттеров и сеттеров (например, могут начинаться с префикса `get/set`, возвращать `void` или некоторый тип и т. д.).

Еще одним вариантом решения является получение всех объявленных полей класса посредством рефлексии (например, посредством метода `Class.getDeclaredFields()`). Теперь обойдем массив `Field[]` в цикле и попытаемся получить геттеры и сеттеры посредством метода `Class.getDeclaredMethod()`, передав ему имя поля (с префиксом `get/set/is` и первой буквой в верхнем регистре) и тип поля (в случае сеттеров).

Наконец, более элегантный вариант решения будет опираться на API `PropertyDescriptor` и API `Introspector`. Оба API доступны в пакете `java.beans.*` и предназначены для работы с `JavaBeans`.



Многие функциональные средства, которые выставляются наружу этими двумя классами, за кулисами основаны на рефлексии.

Класс `PropertyDescriptor` может возвращать метод, используемый для чтения свойства `JavaBean` посредством метода `getReadMethod()`. Кроме того, он может возвращать метод, используемый для записи свойства `JavaBean` посредством метода

`getWriteMethod()`. Опираясь на эти два метода, мы можем получить геттеры и сеттеры класса `Melon` следующим образом:

```
for (PropertyDescriptor pd:
    Introspector.getBeanInfo(Melon.class).getPropertyDescriptors()) {

    if (pd.getReadMethod() != null && !"class".equals(pd.getName())) {
        System.out.println(pd.getReadMethod());
    }

    if (pd.getWriteMethod() != null && !"class".equals(pd.getName())) {
        System.out.println(pd.getWriteMethod());
    }
}
```

Результат таков:

```
public boolean modern.challenge.Melon.isRipe()
public void modern.challenge.Melon.setRipe(boolean)
public java.lang.String modern.challenge.Melon.getType()
public void modern.challenge.Melon.setType(java.lang.String)
public int modern.challenge.Melon.getWeight()
public void modern.challenge.Melon.setWeight(int)
```

Теперь допустим, что у нас есть следующий экземпляр класса `Melon`:

```
Melon melon = new Melon("Gac", 1000);
```

Здесь мы хотим вызвать геттер `getType()`:

```
// возвращаемый тип - Gac
Object type = new PropertyDescriptor("type",
    Melon.class).getReadMethod().invoke(melon);
```

Теперь вызовем сеттер `setweight()`:

```
// установить вес дыни Gac равным 2000
new PropertyDescriptor("weight", Melon.class)
    .getWriteMethod().invoke(melon, 2000);
```

Вызов несуществующего свойства приведет к исключению `IntrospectionException`:

```
try {
    Object shape = new PropertyDescriptor("shape",
        Melon.class).getReadMethod().invoke(melon);
    System.out.println("Форма дыни: " + shape);
} catch (IntrospectionException e) {
    System.out.println("Свойство не найдено: " + e);
}
```

Генерирование геттеров и сеттеров

Допустим, что класс `Melon` имеет три поля (`type`, `weight` и `ripe`) и определяет геттер только для `type` и сеттер только для `ripe`:

```
public class Melon {
    private String type;
    private int weight;
    private boolean ripe;
    ...
    public String getType() {
        return type;
    }

    public void setRipe(boolean ripe) {
        this.ripe = ripe;
    }
    ...
}
```

Для того чтобы сгенерировать недостающие геттеры и сеттеры, мы начнем с их идентификации. Следующее ниже программное решение обходит объявленные поля заданного класса в цикле и исходит из того, что поле `foo` не имеет геттера, если соблюдается следующее:

- ◆ не существует метода `get/isFoo()`;
- ◆ тип возвращаемого значения не совпадает с типом поля;
- ◆ число аргументов не равно 0.

Для каждого отсутствующего геттера это решение добавляет в отображение `Map` элемент, содержащий имя и тип поля:

```
private static Map<String, Class<?>> fetchMissingGetters(Class<?> clazz) {
    Map<String, Class<?>> getters = new HashMap<>();
    Field[] fields = clazz.getDeclaredFields();
    String[] names = new String[fields.length];
    Class<?>[] types = new Class<?>[fields.length];

    Arrays.setAll(names, i -> fields[i].getName());
    Arrays.setAll(types, i -> fields[i].getType());

    for (int i = 0; i < names.length; i++) {
        String getterAccessor = fetchIsOrGet(names[i], types[i]);

        try {
            Method getter = clazz.getDeclaredMethod(getterAccessor);
            Class<?> returnType = getter.getReturnType();
```

```
        if (!returnType.equals(types[i]) ||
            getter.getParameterCount() != 0) {
            getters.put(names[i], types[i]);
        }
    } catch (NoSuchMethodException ex) {
        getters.put(names[i], types[i]);
        // запروتocolировать исключение
    }
}

return getters;
}
```

Далее программное решение обходит объявленные поля заданного класса в цикле и исходит из того, что поле `foo` не имеет сеттера, если соблюдается следующее:

- ◆ поле не является финальным;
- ◆ метода `setFoo()` не существует;
- ◆ метод возвращает `void`;
- ◆ метод имеет один параметр;
- ◆ тип параметра совпадает с типом поля;
- ◆ если имя параметра присутствует, то оно должно совпадать с именем поля.

Для каждого отсутствующего сеттера это решение добавляет элемент, содержащий имя поля и тип, в `map`-отображение:

```
private static Map<String, Class<?>> fetchMissingSetters(Class<?> clazz) {
    Map<String, Class<?>> setters = new HashMap<>();
    Field[] fields = clazz.getDeclaredFields();
    String[] names = new String[fields.length];
    Class<?>[] types = new Class<?>[fields.length];

    Arrays.setAll(names, i -> fields[i].getName());
    Arrays.setAll(types, i -> fields[i].getType());

    for (int i = 0; i < names.length; i++) {
        Field field = fields[i];
        boolean finalField = !Modifier.isFinal(field.getModifiers());

        if (finalField) {
            String setterAccessor = fetchSet(names[i]);

            try {
                Method setter = clazz.getDeclaredMethod(setterAccessor, types[i]);

                if (setter.getParameterCount() != 1 ||
                    !setter.getReturnType().equals(void.class)) {
```

```
        setters.put(names[i], types[i]);
        continue;
    }

    Parameter parameter = setter.getParameters()[0];
    if ((parameter.isNamePresent() &&
        !parameter.getName().equals(names[i])) ||
        !parameter.getType().equals(types[i])) {
        setters.put(names[i], types[i]);
    }
} catch (NoSuchMethodException ex) {
    setters.put(names[i], types[i]);
    // запротоколировать исключение
}
}
}

return setters;
}
```

Так, пока что мы знаем, на каких полях нет геттеров и сеттеров. Их имена и типы хранятся в отображении Map. Давайте обойдем это отображение в цикле и сгенерируем геттеры:

```
public static StringBuilder generateGetters(Class<?> clazz) {
    StringBuilder getterBuilder = new StringBuilder();
    Map<String, Class<?>> accessors = fetchMissingGetters(clazz);

    for (Entry<String, Class<?>> accessor: accessors.entrySet()) {
        Class<?> type = accessor.getValue();
        String field = accessor.getKey();
        String getter = fetchIsOrGet(field, type);

        getterBuilder.append("\npublic ")
            .append(type.getSimpleName()).append(" ")
            .append(getter)
            .append("() {\n")
            .append("\treturn ")
            .append(field)
            .append(";\n")
            .append("}\n");
    }

    return getterBuilder;
}
```


А также сгенерируем сеттеры:

```
public static StringBuilder generateSetters(Class<?> clazz) {
    StringBuilder setterBuilder = new StringBuilder();
    Map<String, Class<?>> accessors = fetchMissingSetters(clazz);

    for (Entry<String, Class<?>> accessor: accessors.entrySet()) {
        Class<?> type = accessor.getValue();
        String field = accessor.getKey();
        String setter = fetchSet(field);

        setterBuilder.append("\npublic void ")
            .append(setter)
            .append("(").append(type.getSimpleName()).append(" ")
            .append(field).append(") {\n")
            .append("\tthis.")
            .append(field).append(" = ")
            .append(field)
            .append(";\n")
            .append("}\n");
    }

    return setterBuilder;
}
```

Приведенное выше решение опирается на троих простых помощников, перечисленных ниже. Исходный код прост и понятен:

```
private static String fetchIsOrGet(String name, Class<?> type) {
    return "boolean".equalsIgnoreCase(type.getSimpleName()) ?
        "is" + uppercase(name) : "get" + uppercase(name);
}

private static String fetchSet(String name) {
    return "set" + uppercase(name);
}

private static String uppercase(String name) {
    return name.substring(0, 1).toUpperCase() + name.substring(1);
}
```

Теперь давайте вызовем его для класса Melon:

```
Class<?> clazz = Melon.class;
StringBuilder getters = generateGetters(clazz);
StringBuilder setters = generateSetters(clazz);
```

На выходе будут показаны следующие сгенерированные геттеры и сеттеры:

```
public int getWeight() {
    return weight;
}
```

```
public boolean isRipe() {  
    return ripe;  
}  
  
public void setWeight(int weight) {  
    this.weight = weight;  
}  
  
public void setType(String type) {  
    this.type = type;  
}
```

158. Рефлексирование аннотаций

Аннотации Java получили много внимания со стороны API рефлексии Java. Рассмотрим решения для проверки нескольких видов аннотаций (например, пакетов, классов и методов).

Все главенствующие классы API рефлексии, представляющие артефакты, поддерживающие аннотации (например, `Package`, `Constructor`, `Class`, `Method` и `Field`), раскрывают набор общих методов для работы с аннотациями. Общие методы включают:

- ◆ `getAnnotations()` — возвращает все аннотации, относящиеся к некоторому артефакту;
- ◆ `getDeclaredAnnotations()` — возвращает все аннотации, объявленные непосредственно к некоторому артефакту;
- ◆ `getAnnotation()` — возвращает аннотацию по типу;
- ◆ `getDeclaredAnnotation()` — возвращает аннотацию по типу, объявленному непосредственно для некоторого артефакта (JDK 1.8);
- ◆ `getDeclaredAnnotationsByType()` — возвращает все аннотации по типу, объявленному непосредственно для некоторого артефакта (JDK 1.8);
- ◆ `isAnnotationPresent()` — возвращает `true`, если на указанном артефакте найдена аннотация для заданного типа.



Метод `getAnnotatedReceiverType()` обсуждался в разд. 152 "Извлечение аннотации типа получателя" ранее в этой главе.

В следующих далее разделах мы поговорим о проверке аннотаций пакетов, классов, методов и т. д.

Инспектирование аннотаций пакетов

Аннотации, специфичные для пакетов, добавляются в `package-info.java`, как показано на рис. 7.4. Здесь пакет `modern.challenge` был помечен аннотацией `@Packt`.

```
@Packt
package modern.challenge;
```

Рис. 7.4

Удобно инспектировать аннотации к пакету, начиная с одного из его классов. Например, если в этом пакете (`modern.challenge`) у нас есть класс `Melon`, мы можем получить все аннотации этого пакета следующим образом:

```
Class<Melon> clazz = Melon.class;
Annotation[] pkggAnnotations = clazz.getPackage().getAnnotations();
```

Массив `Annotation[]`, распечатываемый посредством метода `Arrays.toString()`, показывает один-единственный результат:

```
[@modern.challenge.Packt()]
```

Инспектирование аннотаций классов

Класс `Melon` имеет одну аннотацию — `@Fruit` (рис. 7.5).

```
@Fruit(name = "melon", value = "delicious")
public class Melon extends @Family Cucurbitaceae
    implements @ByWeight Comparable {
```

Рис. 7.5

Но мы можем получить все аннотации посредством метода `getAnnotations()`:

```
Class<Melon> clazz = Melon.class;
Annotation[] clazzAnnotations = clazz.getAnnotations();
```

Возвращаемый массив, распечатываемый посредством метода `Arrays.toString()`, показывает один-единственный результат:

```
[@modern.challenge.Fruit(name="melon", value="delicious")]
```

Для того чтобы получить доступ к атрибутам аннотации, относящимся к имени и значению, мы можем выполнить явное приведение типа:

```
Fruit fruitAnnotation = (Fruit) clazzAnnotations[0];
System.out.println("@Fruit name: " + fruitAnnotation.name());
System.out.println("@Fruit value: " + fruitAnnotation.value());
```

Как вариант, мы можем применить метод `getDeclaredAnnotation()` для прямого получения нужного типа:

```
Fruit fruitAnnotation = clazz.getDeclaredAnnotation(Fruit.class);
```

Инспектирование аннотаций методов

Проинспектируем аннотацию `@Ripe` метода `eat()` из класса `Melon` (рис. 7.6).

Сначала получим все объявленные аннотации, а затем вернемся к `@Ripe`:

```
Class<Melon> clazz = Melon.class;
Method methodEat = clazz.getDeclaredMethod("eat");
Annotation[] methodAnnotations = methodEat.getDeclaredAnnotations();
```

```
@Ripe(true)
public void eat() throws @Runtime IllegalStateException {
}
```

Рис. 7.6

Возвращаемый массив, распечатываемый посредством метода `Arrays.toString()`, показывает один-единственный результат:

```
[@modern.challenge.Ripe(value=true)]
```

И давайте выполним явное приведение типа `methodAnnotations[0]` к `Ripe`:

```
Ripe ripeAnnotation = (Ripe) methodAnnotations[0];
System.out.println("@Ripe value: " + ripeAnnotation.value());
```

Как вариант, мы можем применить метод `getDeclaredAnnotation()` для прямого получения нужного типа:

```
Ripe ripeAnnotation = methodEat.getDeclaredAnnotation(Ripe.class);
```

Инспектирование аннотаций выбрасываемых исключений

Для инспектирования аннотаций выбрасываемых исключений нам нужно вызвать метод `getAnnotatedExceptionTypes()` (рис. 7.7).

```
@Ripe(true)
public void eat() throws @Runtime IllegalStateException {
}
```

Рис. 7.7

Этот метод возвращает типы выбрасываемых исключений, включая те, которые аннотированы:

```
Class<Melon> clazz = Melon.class;
Method methodEat = clazz.getDeclaredMethod("eat");
AnnotatedType[] exceptionsTypes = methodEat.getAnnotatedExceptionTypes();
```

Возвращаемый массив, распечатываемый посредством метода `Arrays.toString()`, показывает один-единственный результат:

```
[@modern.challenge.Runtime() java.lang.IllegalStateException]
```

Получить тип первого исключения можно следующим образом:

```
// class java.lang.IllegalStateException
System.out.println("Тип первого исключения: "
    + exceptionsTypes[0].getType());
```

Получить аннотации типа первого исключения можно так:

```
// [@modern.challenge.Runtime()]
System.out.println("Аннотации типа первого исключения: "
    + Arrays.toString(exceptionsTypes[0].getAnnotations()));
```

Инспектирование аннотаций типов значений, возвращаемых из методов

Для инспектирования аннотаций типа значения, возвращаемого из метода, необходимо вызвать метод `getAnnotatedReturnType()` (рис. 7.8).

```
public @Shape("oval") List<Seed> seeds() {
    return Collections.emptyList();
}
```

Рис. 7.8

Этот метод дает аннотированный тип значения, возвращаемого из заданного метода:

```
Class<Melon> clazz = Melon.class;
Method methodSeeds = clazz.getDeclaredMethod("seeds");
AnnotatedType returnType = methodSeeds.getAnnotatedReturnType();

// java.util.List<modern.Challenge.Seed>
System.out.println("Возвращаемый тип: " + returnType.getType().getTypeName());

// [@modern.challenge.Shape(value="oval")]
System.out.println("Аннотации возвращаемого типа: "
    + Arrays.toString(returnType.getAnnotations()));
```

Инспектирование аннотаций параметров методов

Мы можем проверить аннотации параметров некоторого метода, вызвав метод `getParameterAnnotations()` (рис. 7.9).

```
public void slice(@Ripe(true) @Shape("square") int noOfSlices) {
}
```

Рис. 7.9

Этот метод возвращает матрицу (массив массивов), содержащую аннотации к формальным параметрам, в порядке объявления:

```
Class<Melon> clazz = Melon.class;
Method methodSlice = clazz.getDeclaredMethod("slice", int.class);
Annotation[][] paramAnnotations = methodSlice.getParameterAnnotations();
```

Получить тип каждого параметра с его аннотациями (в данном случае у нас есть параметр типа `int` с двумя аннотациями) можно посредством метода `getParameterTypes()`. Поскольку этот метод также поддерживает порядок объявления, мы можем получить некоторую информацию следующим образом:

```
Class<?>[] parameterTypes = methodSlice.getParameterTypes();

int i = 0;
for (Annotation[] annotations: paramAnnotations) {
```

```
Class parameterType = parameterTypes[i++];
System.out.println("Тип параметра: " + parameterType.getName());

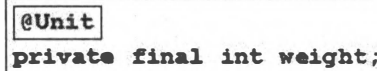
for (Annotation annotation: annotations) {
    System.out.println("Аннотация: " + annotation);
    System.out.println("Имя аннотации: "
        + annotation.annotationType().getSimpleName());
}
}
```

И данные на выходе должны иметь следующий вид:

```
Тип параметра: int
Аннотация: @modern.challenge.Ripe(value=true)
Имя аннотации: Ripe
Аннотация: @modern.challenge.Shape(value="square")
Имя аннотации: Shape
```

Инспектирование аннотаций полей

Имея поле, мы можем получить его аннотации посредством метода `getDeclaredAnnotations()` (рис. 7.10).



```
@Unit
private final int weight;
```

Рис. 7.10

Вот исходный код:

```
Class<Melon> clazz = Melon.class;
Field weightField = clazz.getDeclaredField("weight");
Annotation[] fieldAnnotations = weightField.getDeclaredAnnotations();
```

Получить значение аннотации `@Unit` можно следующим образом:

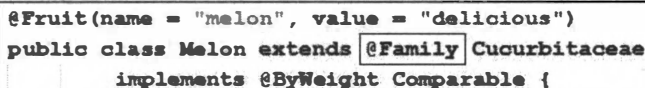
```
Unit unitFieldAnnotation = (Unit) fieldAnnotations[0];
System.out.println("@Unit value: " + unitFieldAnnotation.value());
```

Либо применить метод `getDeclaredAnnotation()`, чтобы получить правильный тип непосредственно:

```
Unit unitFieldAnnotation = weightField.getDeclaredAnnotation(Unit.class);
```

Инспектирование аннотаций надклассов

Для инспектирования аннотаций надкласса необходимо вызвать метод `getAnnotatedSuperclass()` (рис. 7.11).



```
@Fruit(name = "melon", value = "delicious")
public class Melon extends @Family Cucurbitaceae
    implements @ByWeight Comparable {
```

Рис. 7.11

Этот метод возвращает аннотированный тип надкласса:

```
Class<Melon> clazz = Melon.class;
AnnotatedType superclassType = clazz.getAnnotatedSuperclass();
И давайте также получим некоторую информацию:
// modern.challenge.Cucurbitaceae
System.out.println("Тип надкласса: " + superclassType.getType().getTypeName());

// [@modern.challenge.Family()]
System.out.println("Аннотации: "
    + Arrays.toString(superclassType.getDeclaredAnnotations()));

System.out.println("Аннотация @Family присутствует: "
    + superclassType.isAnnotationPresent(Family.class)); // true
```

Инспектирование аннотаций интерфейсов

Для инспектирования аннотаций имплементированных интерфейсов необходимо вызвать метод `getAnnotatedInterfaces()` (рис. 7.12).

```
@Fruit(name = "melon", value = "delicious")
public class Melon extends @Family Cucurbitaceae
    implements @ByWeight Comparable {
```

Рис 7 12

Этот метод возвращает аннотированные типы интерфейсов:

```
Class<Melon> clazz = Melon.class;
AnnotatedType[] interfacesTypes = clazz.getAnnotatedInterfaces();
Возвращаемый массив, распечатываемый посредством метода Arrays.toString(),
показывает один-единственный результат:
```

```
[@modern.challenge.ByWeight() java.lang.Comparable]
Получить тип первого интерфейса можно следующим образом:
```

```
// interface java.lang.Comparable
System.out.println("Тип первого интерфейса: " + interfacesTypes[0].getType());
Кроме того, получить аннотацию типа первого интерфейса можно так:
// [@modern.challenge.ByWeight()]
System.out.println("Аннотации типа первого интерфейса: "
    + Arrays.toString(interfacesTypes[0].getAnnotations()));
```

Получение аннотаций по типу

Имея несколько аннотаций одного и того же типа на некоторых компонентах, мы можем получить их все посредством метода `getAnnotationsByType()`. Для класса мы можем сделать это таким образом:

```
Class<Melon> clazz = Melon.class;
Fruit[] clazzFruitAnnotations = clazz.getAnnotationsByType(Fruit.class);
```

Получение объявленной аннотации

Получить по типу одну аннотацию, объявленную непосредственно на некотором артефакте, можно так, как показано в следующем примере:

```
Class<Melon> clazz = Melon.class;
Method methodEat = clazz.getDeclaredMethod("eat");
Ripe methodRipeAnnotation = methodEat.getDeclaredAnnotation(Ripe.class);
```

159. Инициирование экземплярного метода

Допустим, что мы имеем следующий класс `Melon`:

```
public class Melon {
    ...
    public Melon() {}

    public List<Melon> cultivate(String type, Seed seed, int noOfSeeds) {
        System.out.println("Вызван метод cultivate()...");

        return Collections.nCopies(noOfSeeds, new Melon("Gac", 5));
    }
    ...
}
```

Наша цель — вызвать метод `cultivate()` и получить возвращаемое значение посредством API рефлексии Java.

Сначала получим метод `cultivate()` как объект `Method` посредством метода `Method.getDeclaredMethod()`. Нам нужно лишь передать имя метода (в данном случае `cultivate()`) и правильные типы параметров (`String`, `Seed` и `int`) в метод `getDeclaredMethod()`. Вторым аргументом метода `getDeclaredMethod()` является тип `varargs of Class<?>`, поэтому он может быть пустым для методов без параметров или содержать список типов параметров, как в следующем примере:

```
Method cultivateMethod = Melon.class.getDeclaredMethod(
    "cultivate", String.class, Seed.class, int.class);
```

Затем, получим экземпляр класса `Melon`. Мы хотим вызвать экземплярный метод; следовательно, нам нужен экземпляр. Опираясь на пустой конструктор класса `Melon` и API рефлексии Java, мы можем сделать это следующим образом:

```
Melon instanceMelon = Melon.class
    .getDeclaredConstructor().newInstance();
```

Наконец, мы сосредоточимся на методе `Method.invoke()`. В общих чертах, мы должны передать этому методу экземпляр, используемый для вызова метода `cultivate()`, и некоторые значения для параметров:

```
List<Melon> cultivatedMelons = (List<Melon>) cultivateMethod.invoke(
    instanceMelon, "Gac", new Seed(), 10);
```


Успех вызова проявляется следующим сообщением:

Вызван метод cultivate()...

Более того, если мы напечатаем данные, возвращаемые в результате вызова, посредством инструкции `System.out.println()`, то получим такой результат:

```
[Gac(5g), Gac(5g), Gac(5g), ...]
```

Мы только что выполнили культивацию 10 дынь посредством рефлексии.

160. Получение статических методов

Допустим, что мы имеем следующий класс `Melon`:

```
public class Melon {  
    ...  
    public void eat() {}  
  
    public void weighsIn() {}  
  
    public static void cultivate(Seed seeds) {  
        System.out.println("Вызван метод cultivate()...");  
    }  
  
    public static void peel(Slice slice) {  
        System.out.println("Вызван метод peel()...");  
    }  
  
    // геттеры, сеттеры, toString() опущены для краткости  
}
```

Этот класс имеет два статических метода — `cultivate()` и `peel()`. Давайте извлечем оба метода в список `List<Method>`.

Решение этой задачи состоит из двух главных шагов:

1. Получить все имеющиеся методы заданного класса.
2. Отфильтровать те, которые содержат модификатор `static`, посредством метода `Modifier.isStatic()`.

В исходном коде это выглядит так:

```
List<Method> staticMethods = new ArrayList<>();  
  
Class<Melon> clazz = Melon.class;  
Method[] methods = clazz.getDeclaredMethods();  
  
for (Method method: methods) {  
    if (Modifier.isStatic(method.getModifiers())) {  
        staticMethods.add(method);  
    }  
}
```

Результат печати списка с помощью инструкции `System.out.println()` выглядит следующим образом:

```
[public static void
  modern.challenge.Melon.peel(modern.challenge.Slice),

  public static void
    modern.challenge.Melon.cultivate(modern.challenge.Seed)]
```

Еще один шаг, и мы, возможно, захотим вызвать один из этих двух методов.

Например, вызовем метод `peel()` (обратите внимание, что мы передаем `null` вместо экземпляра класса `Melon`, т. к. статический метод не нуждается в экземпляре):

```
Method method = clazz.getMethod("peel", Slice.class);
method.invoke(null, new Slice());
```

Выходные данные сигнализируют о том, что метод `peel()` был успешно вызван:

Вызван метод `peel()`...

161. Получение обобщенных методов, полей и исключений

Допустим, что мы имеем следующий ниже класс `Melon` (перечислены только те части, которые имеют отношение к этой задаче):

```
public class Melon<E extends Exception>
    extends Fruit<String, Seed> implements Comparable<Integer> {

    ...
    private List<Slice> slices;
    ...

    public List<Slice> slice() throws E {
        ...
    }

    public Map<String, Integer> asMap(List<Melon> melons) {
        ...
    }
    ...
}
```

Класс `Melon` содержит несколько обобщенных типов, ассоциированных с разными артефактами. Обобщенные типы надклассов, интерфейсов, классов, методов и полей являются экземплярами класса `ParameterizedType`. Для каждого параметризованного типа нам нужно получить фактический тип аргументов посредством метода `ParameterizedType.getActualTypeArguments()`. Массив `Type[]`, возвращаемый этим ме-

тодом, может быть обойден в цикле для получения информации о каждом аргументе следующим образом:

```
public static void printGenerics(Type genericType) {
    if (genericType instanceof ParameterizedType) {
        ParameterizedType type = (ParameterizedType) genericType;

        Type[] typeOfArguments = type.getActualTypeArguments();

        for (Type typeOfArgument: typeOfArguments) {
            Class classTypeOfArgument = (Class) typeOfArgument;
            System.out.println("Класс типа аргумента: " + classTypeOfArgument);

            System.out.println("Простое имя типа аргумента: "
                + classTypeOfArgument.getSimpleName());
        }
    }
}
```

Теперь давайте посмотрим, как работать с обобщенными методами.

Обобщения методов

Например, получим обобщенные типы возвращаемых данных для методов `slice()` и `asMap()`. Это может быть сделано посредством метода `Method.getGenericReturnType()` следующим образом:

```
Class<Melon> clazz = Melon.class;

Method sliceMethod = clazz.getDeclaredMethod("slice");
Method asMapMethod = clazz.getDeclaredMethod("asMap", List.class);

Type sliceReturnType = sliceMethod.getGenericReturnType();
Type asMapReturnType = asMapMethod.getGenericReturnType();
```

Теперь вызов метода `printGenerics(sliceReturnType)` напечатает следующее:

```
Класс типа аргумента: class modern.challenge.Slice
Простое имя типа аргумента: Slice
```

И вызов метода `printGenerics(asMapReturnType)` напечатает следующее:

```
Класс типа аргумента: class java.lang.String
Простое имя типа аргумента: String
```

```
Класс типа аргумента: class java.lang.Integer
Простое имя типа аргумента: Integer
```

Обобщенные параметры методов могут быть получены посредством метода `Method.getGenericParameterTypes()` следующим образом:

```
Type[] asMapParamTypes = asMapMethod.getGenericParameterTypes();
```

Далее мы вызываем метод `printGenerics()` для каждого типа (каждого обобщенного параметра):

```
for (Type paramType: asMapParamTypes) {  
    printGenerics(paramType);  
}
```

Ниже приведен результат (с одним-единственным обобщенным параметром `List<Melon>`):

Класс типа аргумента: `class modern.challenge.Melon`

Простое имя типа аргумента: `Melon`

Обобщения полей

В случае полей (например, `slices`) обобщения могут быть получены посредством метода `Field.getGenericType()` так:

```
Field slicesField = clazz.getDeclaredField("slices");  
Type slicesType = slicesField.getGenericType();
```

Вызов метода `printGenerics(slicesType)` приведет к следующему результату:

Класс типа аргумента: `class modern.challenge.Slice`

Простое имя типа аргумента: `Slice`

Обобщения надклассов

Получить обобщение надкласса можно посредством метода `getGenericSuperclass()` текущего класса:

```
Type superclassType = clazz.getGenericSuperclass();
```

Вызов метода `printGenerics(superclassType)` приведет к следующему результату:

Класс типа аргумента: `class java.lang.String`

Простое имя типа аргумента: `String`

Класс типа аргумента: `class modern.challenge.Seed`

Простое имя типа аргумента: `Seed`

Обобщения интерфейсов

Получить обобщения имплементированных интерфейсов можно посредством метода `getGenericInterfaces()` текущего класса:

```
Type[] interfacesTypes = clazz.getGenericInterfaces();
```

Далее мы вызываем метод `printGenerics()` для каждого типа. Ниже приведен результат (имеется один-единственный интерфейс, `Comparable<Integer>`):

Класс типа аргумента: `class java.lang.Integer`

Простое имя типа аргумента: `Integer`

Обобщения исключений

Обобщенные типы исключений материализуются в экземплярах классов `TypeVariable` или `ParameterizedType`. На этот раз вспомогательный метод получения и

печати информации об обобщениях, основываясь на классе `TypeVariable`, можно записать следующим образом:

```
public static void printGenericsOfExceptions(Type genericType) {
    if (genericType instanceof TypeVariable) {
        TypeVariable typeVariable = (TypeVariable) genericType;
        GenericDeclaration genericDeclaration
            = typeVariable.getGenericDeclaration();

        System.out.println("Обобщенное объявление: " + genericDeclaration);

        System.out.println("Границы: ");
        for (Type type: typeVariable.getBounds()) {
            System.out.println(type);
        }
    }
}
```

Имея этого помощника, мы можем передавать ему исключения, выбрасываемые посредством метода `getGenericExceptionTypes()`. Если тип исключения представлен переменной типа (`TypeVariable`) или параметризованным типом (`ParameterizedType`), то оно создается. В противном случае оно обрабатывается:

```
Type[] exceptionsTypes = sliceMethod.getGenericExceptionTypes();
```

Далее мы вызываем метод `printGenerics()` для каждого типа:

```
for (Type paramType: exceptionsTypes) {
    printGenericsOfExceptions(paramType);
}
```

Результат будет следующим:

Обобщенное объявление: `class modern.challenge.Melon`

Границы: `class java.lang.Exception`



Скорее всего, печать полученной информации об обобщениях не будет полезной, поэтому смело можете переделать приведенные выше помощники, исходя из ваших потребностей. Например, соберите информацию и верните ее в виде списка, отображения и т. д.

162. Получение публичных и частных полей

Решение этой задачи опирается на флаговые методы `Modifier.isPublic()` и `Modifier.isPrivate()`.

Допустим, что следующий ниже класс `Melon` имеет два публичных поля и два частных поля:

```
public class Melon {
    private String type;
    private int weight;
```

```
public Peeler peeler;
public Juicer juicer;
...
}
```

Сначала нам нужно получить массив `Field[]`, соответствующий этому классу, посредством метода `getDeclaredFields()`:

```
Class<Melon> clazz = Melon.class;
Field[] fields = clazz.getDeclaredFields();
```

Массив `Field[]` содержит все четыре поля из указанных ранее. Далее обойдем этот массив в цикле и применим флаговые методы `Modifier.isPublic()` и `Modifier.isPrivate()` для каждого поля:

```
List<Field> publicFields = new ArrayList<>();
List<Field> privateFields = new ArrayList<>();

for (Field field: fields) {
    if (Modifier.isPublic(field.getModifiers())) {
        publicFields.add(field);
    }

    if (Modifier.isPrivate(field.getModifiers())) {
        privateFields.add(field);
    }
}
```

Список `publicFields` содержит только публичные поля, а список `privateFields` только приватные поля. Если мы напечатаем оба списка с помощью инструкции `System.out.println()`, то результат будет следующим:

```
Public fields:
[public modern.challenge.Peeler modern.challenge.Melon.peeler,
public modern.challenge.Juicer modern.challenge.Melon.juicer]
```

```
Private fields:
[private java.lang.String modern.challenge.Melon.type,
private int modern.challenge.Melon.weight]
```

163. Работа с массивами

API рефлексии Java идет в комплекте с классом, предназначенным для работы с массивами. Этот класс называется `java.lang.reflect.Array`.

Например, следующий ниже фрагмент кода создает массив значений типа `int`. Первый параметр указывает, какого типа должен быть каждый элемент в массиве. Второй параметр представляет длину массива. Таким образом, массив из 10 целых чисел может быть определен посредством метода `Array.newInstance()`:

```
int[] arrayOfInt = (int[]) Array.newInstance(int.class, 10);
```

Используя рефлексию Java, мы можем изменять содержимое массива. Существуют общий метод `set()` и связка методов `setFoo()` (например, `setInt()` и `setFloat()`). Установить значение в индексе от 0 до 100 можно следующим образом:

```
Array.setInt(arrayOfInt, 0, 100);
```

Получить значения из массива можно посредством методов `get()` и `getFoo()` (эти методы получают массив и индекс в качестве аргументов и возвращают значение в соответствии с указанным индексом):

```
int valueIndex0 = Array.getInt(arrayOfInt, 0);
```

Получить Class массива можно следующим образом:

```
Class<?> stringClass = String[].class;
```

```
Class<?> clazz = arrayOfInt.getClass();
```

И мы можем получить тип массива посредством `getComponentType()`:

```
// int
```

```
Class<?> typeInt = clazz.getComponentType();
```

```
// java.lang.String
```

```
Class<?> typeString = stringClass.getComponentType();
```

164. Инспектирование модулей

Java 9 добавил понятие *модулей* посредством модульной системы платформы Java. В сущности, модуль — это совокупность пакетов, управляемых этим модулем (например, модуль решает, какие пакеты видны за пределами модуля).

Приложение с двумя модулями может быть сформировано так, как показано на рис. 7.13.

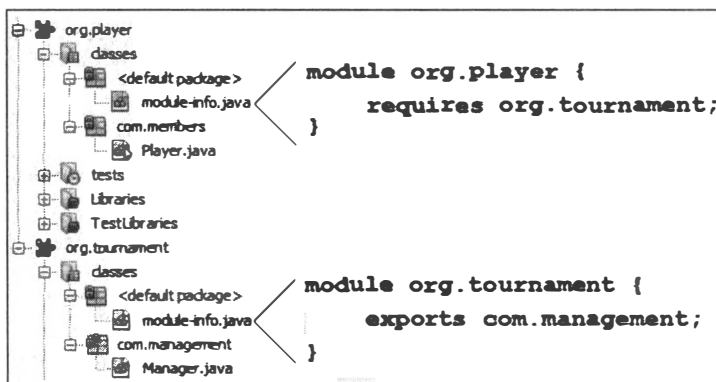


Рис. 7.13

Имеются два модуля — `org.player` и `org.tournament`. Модуль `org.player` требует наличия модуля `org.tournament`, модуль `org.tournament` экспортирует пакет `com.management`.

API рефлексии Java представляет модуль посредством класса `java.lang.Module` (в модуле `java.base`). С помощью API рефлексии Java мы можем получать информацию или модифицировать модуль.

Для начала мы можем получить экземпляр класса `Module`, как показано в следующих двух примерах:

```
Module playerModule = Player.class.getModule();
Module managerModule = Manager.class.getModule();
```

Имя модуля можно получить посредством метода `Module.getName()`:

```
// org.player
System.out.println("Класс 'Player' находится в модуле: "
    + playerModule.getName());

// org.tournament
System.out.println("Класс 'Manager' находится в модуле: "
    + managerModule.getName());
```

Имея экземпляр класса `Module`, мы можем вызвать несколько методов для получения разной информации. Например, мы можем узнать, существует ли имя у модуля, либо экспортировал или открыл ли модуль некоторый пакет:

```
boolean playerModuleIsNamed = playerModule.isNamed(); // true
boolean managerModuleIsNamed = managerModule.isNamed(); // true

boolean playerModulePnExported
    = playerModule.isExported("com.members"); // false
boolean managerModulePnExported
    = managerModule.isExported("com.management"); // true

boolean playerModulePnOpen
    = playerModule.isOpen("com.members"); // false
boolean managerModulePnOpen
    = managerModule.isOpen("com.management"); // false
```

Помимо получения информации, класс `Module` позволяет нам изменять модуль. Например, модуль `org.player` не экспортирует пакет `com.members` в модуль `org.tournament`. Мы можем это быстро проверить:

```
boolean before = playerModule.isExported(
    "com.members", managerModule); // false
```

Но можно изменить это путем рефлексии. Мы можем выполнить этот экспорт посредством метода `Module.addExports()` (в той же категории у нас есть методы `addOpens()`, `addReads()` и `addUses()`):

```
playerModule.addExports("com.members", managerModule);
```

А теперь давайте проверим еще раз:

```
boolean after = playerModule.isExported("com.members", managerModule); // true
```


Модуль также использует преимущества собственного дескриптора. Класс `ModuleDescriptor` можно использовать в качестве отправной точки для работы с модулем:

```
ModuleDescriptor descriptorPlayerModule = playerModule.getDescriptor();
```

Например, мы можем получить пакеты модуля следующим образом:

```
Set<String> pcks = descriptorPlayerModule.packages();
```

165. Динамические посредники

Динамические посредники (прокси) используются для поддержки имплементации разных функциональностей, входящих в категорию межобъектных (сквозных) обязанностей (cross cutting-concerns, CCC). **Межобъектные обязанности** — это те задачи, которые представляют собой вспомогательные функциональности ключевых функциональностей, такие как управление подключением к базе данных, управление транзакциями (например, `@Transactional` в прикладном каркасе Spring), безопасность и журналирование/протоколирование.

Точнее, рефлексия Java идет в комплекте с классом под названием `java.lang.reflect.Proxy`, главной целью которого является обеспечение поддержки создания динамических имплементаций интерфейсов во время выполнения. Класс `Proxy` отражает имплементацию конкретного интерфейса во время выполнения.

Мы можем думать о `Proxy` как о *фронтальной обертке*, которая передает наши вызовы нужным методам. Дополнительно `Proxy` может вмешиваться в процесс перед делегированием вызова.

Динамические посредники опираются на один-единственный класс (`InvocationHandler`) с одним-единственным методом (`invoke()`), как показано на рис. 7.14.

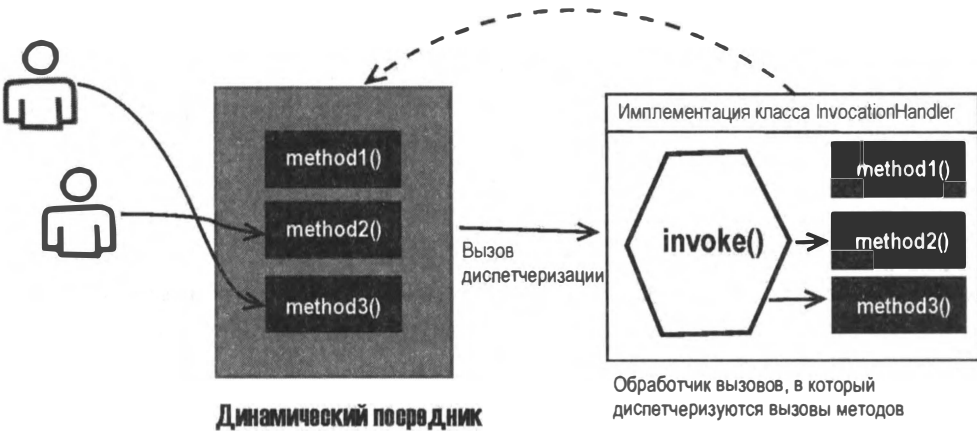


Рис. 7 14

Если мы изобразим поток управления на этой диаграмме, то получим следующие шаги:

1. Акторы (actors) вызывают необходимые методы через выставленного наружу динамического посредника (например, если мы хотим вызвать метод `List.add()`, то будем делать это через динамического посредника, а не напрямую).
2. Динамический посредник диспетчеризует вызов экземпляру имплементации класса `InvocationHandler` (с каждым экземпляром посредника ассоциирован обработчик вызовов).
3. Диспетчеризованный вызов попадает в метод `invoke()` как триада, содержащая объект посредника, вызываемый метод (как экземпляр класса `Method`) и массив аргументов для этого метода.
4. Экземпляр класса `InvocationHandler` выполнит дополнительные необязательные функциональности (например, межобъектные обязанности) и вызовет соответствующий метод.
5. Экземпляр класса `InvocationHandler` вернет результат вызова в качестве объекта.

Если мы попытаемся резюмировать этот процесс, то можно сказать, что динамический посредник поддерживает вызовы нескольких методов произвольных классов посредством одного класса (`InvocationHandler`) с одним методом (`invoke()`).

Имплементирование динамического посредника

Давайте напишем динамического посредника, который подсчитывает число вызовов методов класса `List`.

Динамический посредник создается посредством метода `Proxy.newProxyInstance()`. Метод `newProxyInstance()` берет три параметра:

- ◆ `ClassLoader` используется для загрузки класса динамического посредника;
- ◆ `Class<?>[]` — это массив имплементируемых интерфейсов;
- ◆ `InvocationHandler` — это обработчик вызовов, в который диспетчеризируются вызовы методов.

Взгляните вот на этот пример:

```
List<String> listProxy = (List<String>) Proxy.newProxyInstance(
    List.class.getClassLoader(), new Class[] {
        List.class}, invocationHandler);
```

Указанный фрагмент кода возвращает динамическую имплементацию интерфейса `List`. Далее все вызовы через этого посредника будут диспетчеризоваться в экземпляр класса `invocationHandler`.

В общих чертах скелет имплементации класса `InvocationHandler` выглядит следующим образом:

```
public class DummyInvocationHandler implements InvocationHandler {
    @Override
```

```
public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    ...
}
```

Поскольку мы хотим подсчитать число вызовов методов класса `List`, мы должны хранить сигнатуры всех методов и число вызовов каждого из них. Это можно выполнить посредством отображения `Map`, инициализированного в конструкторе класса `CountingInvocationHandler` следующим образом (он является нашей имплементацией класса `InvocationHandler`, и `invocationHandler` — его экземпляр):

```
public class CountingInvocationHandler implements InvocationHandler {
    private final Map<String, Integer> counter = new HashMap<>();
    private final Object targetObject;

    public CountingInvocationHandler(Object targetObject) {
        this.targetObject = targetObject;

        for (Method method:targetObject.getClass().getDeclaredMethods()) {
            this.counter.put(method.getName()
                + Arrays.toString(method.getParameterTypes()), 0);
        }
    }
    ...
}
```

Поле `targetObject` содержит имплементацию интерфейса `List` (в данном случае `ArrayList`).

И мы создаем экземпляр класса `CountingInvocationHandler` следующим образом:

```
CountingInvocationHandler invocationHandler
    = new CountingInvocationHandler(new ArrayList<>());
```

Метод `invoke()` просто подсчитывает вызовы и вызывает метод с указанными аргументами:

```
@Override
public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    Object resultOfInvocation = method.invoke(targetObject, args);
    counter.computeIfPresent(method.getName()
        + Arrays.toString(method.getParameterTypes()), (k, v) -> ++v);

    return resultOfInvocation;
}
```

Наконец, мы выставляем наружу метод, который возвращает число вызовов заданного метода:

```
public Map<String, Integer> countOf(String methodName) {
    Map<String, Integer> result = counter.entrySet().stream()
```

```
.filter(e -> e.getKey().startsWith(methodName + "["))
.filter(e -> e.getValue() != 0)
.collect(Collectors.toMap(Entry::getKey, Entry::getValue));

return result;
}
```

Исходный код, прилагаемый к этой книге, склеивает все эти фрагменты кода вместе в классе с именем `CountingInvocationHandler`.

На данный момент мы можем использовать `listProxy` для вызова нескольких методов:

```
listProxy.add("Adda");
listProxy.add("Mark");
listProxy.add("John");
listProxy.remove("Adda");
listProxy.add("Marcel");
listProxy.remove("Mark");
listProxy.add(0, "Akiuy");
```

И давайте посмотрим, сколько раз мы вызывали методы `add()` и `remove()`:

```
// {add[class java.lang.Object]=4, add[int, class java.lang.Object]=1}
invocationHandler.countOf("add");

// {remove[class java.lang.Object]=2}
invocationHandler.countOf("remove");
```



Поскольку метод `add()` вызывался посредством двух своих сигнатур, результирующее отображение `Map` содержит два элемента.

Резюме

Ну вот, рассмотрена последняя задача в настоящей главе. Надо надеяться, мы в полной мере провели всеобъемлющий обзор API рефлексии Java. Мы подробно рассмотрели задачи, касающиеся классов, интерфейсов, конструкторов, методов, полей, аннотаций и т. д.

Вы можете скачать приложения из этой главы, чтобы увидеть результаты и получить дополнительные детали, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

8

Программирование в функциональном стиле — основы и шаблоны архитектурного дизайна

Эта глава содержит 11 задач с привлечением программирования на Java в функциональном стиле. Мы начнем с задачи, которая призвана обеспечить исчерпывающую информацию о переходе от кода без интерфейсов к коду с функциональными интерфейсами. Затем мы рассмотрим набор шаблонов архитектурного дизайна из книги "банды четырех" (GoF), которые мы проинтерпретируем в функциональном стиле Java.

К концу этой главы вы познакомитесь с функциональным стилем программирования и будете готовы продолжить работу с набором задач, которые позволят нам глубоко погрузиться в эту тему. Вы должны уметь применять ряд часто используемых шаблонов архитектурного дизайна, написанных в функциональном стиле, и очень хорошо понимать, как разрабатывать код с учетом преимуществ функциональных интерфейсов.

Задачи

Используйте следующие задачи для проверки вашего умения программировать в функциональном стиле. Перед тем как обратиться к решениям и скачать примеры программ, настоятельно рекомендуется изучить каждую задачу:

166. **Написание функциональных интерфейсов.** Написать программу, показывающую путь от 0 интерфейсов до функционального интерфейса с помощью набора содержательных примеров.
167. **Лямбда-выражение в двух словах.** Объяснить, что такое лямбда-выражение.

168. **Имплементирование шаблона исполнения вокруг опорной операции.** Написать программу, которая является имплементацией шаблона исполнения вокруг опорной операции на основе лямбда-выражений.
169. **Имплементирование шаблона фабрики.** Написать программу, которая является имплементацией шаблона фабрики на основе лямбда-выражений.
170. **Имплементирование шаблона стратегий.** Написать программу, которая является имплементацией шаблона стратегий на основе лямбда-выражений.
171. **Имплементирование шаблона трафаретного метода.** Написать программу, которая является имплементацией шаблона трафаретного метода на основе лямбда-выражений.
172. **Имплементирование шаблона наблюдателя.** Написать программу, которая является имплементацией шаблона наблюдателя на основе лямбда-выражений.
173. **Имплементирование шаблона одалживания.** Написать программу, которая является имплементацией шаблона одалживания на основе лямбда-выражений.
174. **Имплементирование шаблона декоратора.** Написать программу, которая является имплементацией шаблона декоратора на основе лямбда-выражений.
175. **Имплементирование шаблона каскадного строителя.** Написать программу, которая является имплементацией шаблона каскадного строителя на основе лямбда-выражений.
176. **Имплементирование шаблона команд.** Написать программу, которая является имплементацией шаблона команд на основе лямбда-выражений.

Решения

В следующих разделах описаны решения перечисленных выше задач. Помните, что обычно не существует одного единственно правильного варианта решения той или иной задачи. Кроме того, имейте в виду, что приведенные здесь объяснения включают только самые интересные и важные детали, необходимые для решения. Вы можете скачать примеры решений, чтобы увидеть дополнительные детали и поэкспериментировать с программами, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

166. Написание функциональных интерфейсов

В решении этой задачи мы высветим предназначение и удобство использования функционального интерфейса по сравнению с несколькими альтернативами. Мы посмотрим на то, как эволюционировать код, начав с его базовой и жесткой имплементации в сторону гибкой имплементации, основанной на функциональном интерфейсе.

Для этого рассмотрим следующий класс `Melon`:

```
public class Melon {
    private final String type;
    private final int weight;
    private final String origin;

    public Melon(String type, int weight, String origin) {
        this.type = type;
        this.weight = weight;
        this.origin = origin;
    }

    // геттеры, toString() и другие методы опущены для краткости
}
```

Допустим, что у нас есть клиент — назовем его Марком, — который хочет открыть свое дело по продаже дынь. Мы сформировали предыдущий класс, основываясь на предоставленном им описании. Его главная цель — иметь приложение учета запасов, которое будет поддерживать его идеи и решения, поэтому необходимо создать приложение, которое должно расти вместе с потребностями бизнеса и эволюции. В следующих далее разделах мы оценим время, необходимое для ежедневной разработки этого приложения.

День 1 (фильтрация дынь по их сорту)

В один из дней Марк попросил нас предоставить функцию для фильтрации дынь по их сорту. В результате мы создали служебный класс с именем `Filters` и имплементировали статический метод, который в качестве аргументов берет список дынь и фильтруемый сорт.

Полученный метод довольно прост:

```
public static List<Melon> filterByType(List<Melon> melons, String type) {
    List<Melon> result = new ArrayList<>();

    for (Melon melon: melons) {
        if (melon != null && type.equalsIgnoreCase(melon.getType())) {
            result.add(melon);
        }
    }

    return result;
}
```

Готово! Теперь мы можем легко фильтровать дыни по сорту, как показано в следующем примере:

```
List<Melon> bailans = Filters.filterByType(melons, "Bailan");
```

День 2 (фильтрация дынь по их весу)

Хотя Марк был удовлетворен результатом, он попросил еще один фильтр для получения дынь того или иного веса (например, всех дынь, которые весят 1200 г). Мы только что имплементировали фильтр подобного рода для сортов дынь, и поэтому мы можем придумать новый статический метод для дынь того или иного веса, как показано ниже:

```
public static List<Melon> filterByWeight(List<Melon> melons, int weight) {
    List<Melon> result = new ArrayList<>();

    for (Melon melon: melons) {
        if (melon != null && melon.getWeight() == weight) {
            result.add(melon);
        }
    }

    return result;
}
```

Этот код похож на метод `filterByType()`, за исключением того, что он имеет другое условие/фильтр. Как разработчики, мы начинаем понимать, что если мы будем продолжать в том же духе, то класс `Filters` в итоге получит много методов, которые просто повторяют код и используют лишь другое условие. Здесь мы очень близки к случаю, когда имеется *стереотипный код*.

День 3 (фильтрация дынь по их сорту и весу)

Все становится еще хуже. Теперь Марк попросил нас добавить новый фильтр, который фильтрует дыни по сорту и весу, и ему нужно, чтобы тот был очень быстрым. Однако самая быстрая имплементация является самой уродливой. Взгляните на вот это:

```
public static List<Melon> filterByTypeAndWeight(
    List<Melon> melons, String type, int weight) {

    List<Melon> result = new ArrayList<>();

    for (Melon melon: melons) {
        if (melon != null && type.equalsIgnoreCase(melon.getType())
            && melon.getWeight() == weight) {
            result.add(melon);
        }
    }

    return result;
}
```


В нашем контексте это неприемлемо. Если мы добавим сюда новый критерий фильтра, то код станет трудно обслуживать технически, а также он будет подвержен ошибкам.

День 4 (передача поведения в качестве параметра)

И вот опять! Бессмысленно дальше добавлять фильтры подобного рода. Фильтрация с каждым новым придуманным атрибутом в итоге приведет к огромному классу `Filters`, в котором будут громоздкие, сложные методы со слишком большим числом параметров и тоннами *стереотипного* кода.

Главная проблема заключается в том, что у нас разные образцы поведения завернуты в *стереотипный* код. Поэтому будет неплохо написать стереотипный код только один раз и передавать в него поведение в качестве параметра. Благодаря этому мы можем формировать любые условия/критерии отбора как поведение и жонглировать ими по своему желанию. Код станет понятнее, гибче, проще в сопровождении и будет иметь меньше параметров.

Этот подход называется **параметризацией поведения**, которая представлена на рис. 8.1 (слева показано то, что мы имеем сейчас; справа — то, что мы хотим получить).

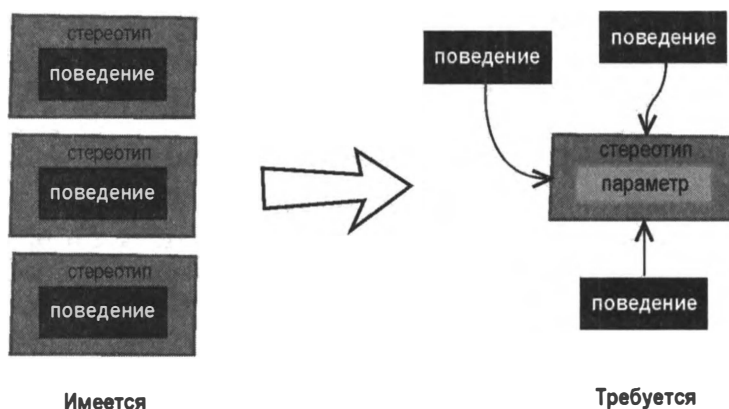


Рис. 8.1

Если мы подумаем о каждом условии/критерии отбора как о поведении, то интуитивно так и напрашивается рассматривать каждый образец поведения как имплементацию интерфейса. В сущности, все эти линии поведения имеют нечто общее — условие/критерий отбора и возвращаемое значение типа `boolean` (то, что мы называем **предикатом**). В контексте интерфейса это представляет собой контракт, который можно написать следующим образом:

```
public interface MelonPredicate {
    boolean test(Melon melon);
}
```

Далее мы можем написать разные имплементации интерфейса `MelonPredicate`. Например, фильтрация дыни `Gac` может быть написана следующим образом:

```
public class GacMelonPredicate implements MelonPredicate {
    @Override
    public boolean test(Melon melon) {
        return "gac".equalsIgnoreCase(melon.getType());
    }
}
```

В качестве альтернативы фильтрация всех дынь, которые тяжелее 5000 г, может быть написана как:

```
public class HugeMelonPredicate implements MelonPredicate {
    @Override
    public boolean test(Melon melon) {
        return melon.getWeight() > 5000;
    }
}
```

Этот технический прием имеет название "шаблон стратегий". Согласно "банде четырех", указанный шаблон "определяет семейство алгоритмов (стратегии), инкапсулирует каждый из них и делает их взаимозаменяемыми. Шаблон стратегий позволяет алгоритму варьироваться независимо от клиента к клиенту".

Итак, главная идея заключается в динамическом отборе поведения алгоритма во время выполнения. Интерфейс `MelonPredicate` объединяет все алгоритмы, предназначенные для отбора дынь, и каждая его имплементация является стратегией.

На данный момент у нас есть стратегии, но у нас нет метода, который получает параметр `MelonPredicate`. Нам нужен метод `filterMelons()`, как показано на рис. 8.2.

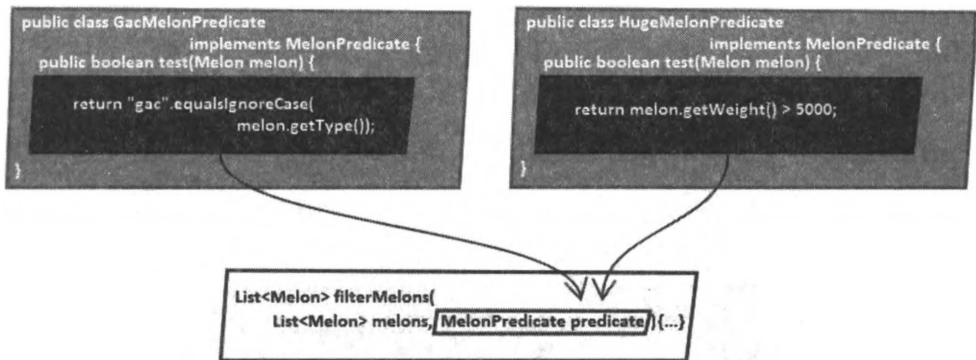


Рис. 8.2

Итак, нам нужны один параметр и несколько поведений. Давайте посмотрим на исходный код метода `filterMelons()`:

```
public static List<Melon> filterMelons(
    List<Melon> melons, MelonPredicate predicate) {
```

```

List<Melon> result = new ArrayList<>();

for (Melon melon: melons) {
    if (melon != null && predicate.test(melon)) {
        result.add(melon);
    }
}

return result;
}

```

Он выглядит гораздо лучше! Мы можем использовать этот метод повторно с разными поведением следующим образом (здесь мы передаем `GacMelonPredicate` и `HugeMelonPredicate`):

```

List<Melon> gacs = Filters.filterMelons(melons, new GacMelonPredicate());

List<Melon> huge = Filters.filterMelons(melons, new HugeMelonPredicate());

```

День 5 (имплементирование еще 100 фильтров)

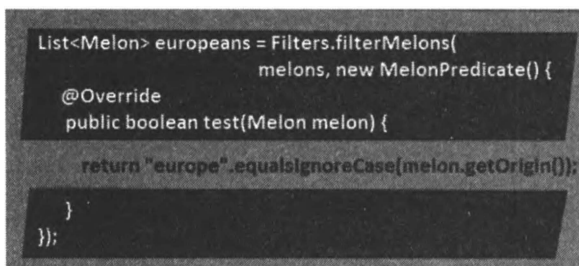
Марк попросил нас имплементировать еще 100 фильтров. На этот раз у нас есть гибкость и поддержка для выполнения этой операции, но нам еще остается написать 100 стратегий или классов для имплементации интерфейса `MelonPredicate` для каждого критерия отбора. Кроме того, мы должны создать экземпляры этих стратегий и передать их в метод `filterMelons()`.

Это означает много кода и времени. Для того чтобы сэкономить и на том и на другом, мы можем опереться на анонимные классы Java. Другими словами, наличие безымянных классов, которые объявляются и инстанцируются в одно и то же время, приведет к чему-то вроде вот этого:

```

List<Melon> europeans = Filters.filterMelons(melons, new MelonPredicate() {
    @Override
    public boolean test(Melon melon) {
        return "europe".equalsIgnoreCase(melon.getOrigin());
    }
});

```



```

List<Melon> europeans = Filters.filterMelons(
    melons, new MelonPredicate() {
        @Override
        public boolean test(Melon melon) {
            return "europe".equalsIgnoreCase(melon.getOrigin());
        }
    });

```

Рис. 8.3

Здесь есть некоторый прогресс, но он не очень существенен, потому что нам все еще нужно написать много кода. Взгляните на выделенный код на рис. 8.3 (этот код повторяется для каждого имплементированного поведения).

Здесь код не является дружелюбным. Анонимные классы кажутся сложными, и они почему-то выглядят неполными и странными, в особенности для новичков.

День 6 (анонимные классы могут быть написаны как лямбда-выражения)

Новый день, новая идея! Любой умный IDE может указать нам дорогу. Например, IDE NetBeans дискретно предупредит нас о том, что этот анонимный класс можно написать как лямбда-выражение.

Это показано на рис. 8.4.

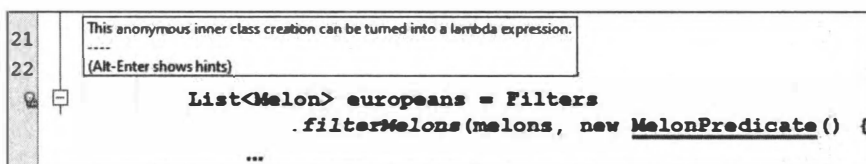


Рис. 8.4

Сообщение является кристально ясным — *это создание анонимного внутреннего класса может быть преобразовано в лямбда-выражение* (This anonymous inner class creation can be turned into a lambda expression). Здесь можно сделать преобразование вручную либо дать возможность IDE сделать это за нас.

Результат будет выглядеть так:

```
List<Melon> europeansLambda = Filters.filterMelons(  
    melons, m -> "europe".equalsIgnoreCase(m.getOrigin()));
```

Данный вариант выглядит гораздо лучше! На этот раз лямбда-выражения Java 8 проделали отличную работу. Теперь мы можем писать фильтры Марка гибче, быстрее, чище, в более удобочитаемом и технически сопровождаемом виде.

День 7 (абстрагирование от типа *List*)

На следующий день Марк приходит с хорошими новостями — он собирается расширить свой бизнес и будет продавать другие фрукты, помимо дынь. Это круто, но наш предикат поддерживает только экземпляры класса `Melon`.

Итак, каким же образом мы должны перейти к тому, чтобы поддерживать и другие плоды? Сколько же еще этих фруктов? Что, если Марк решит начать продавать другую категорию продуктов, например овощи? Мы не можем просто взять и создавать предикат для каждого из них. Это вернет нас к началу.

Очевидным решением является абстрагирование от типа `List`. Начнем с того, что определим новый интерфейс, и на этот раз назовем его `Predicate` (удалив `Melon` из имени):

```
@FunctionalInterface
```

```
public interface Predicate<T> {  
    boolean test(T t);  
}
```

Далее мы перепишем метод `filterMelons()` и переименуем его в `filter()`:

```
public static <T> List<T> filter(List<T> list, Predicate<T> predicate) {  
    List<T> result = new ArrayList<>();  
  
    for (T t: list) {  
        if (t != null && predicate.test(t)) {  
            result.add(t);  
        }  
    }  
  
    return result;  
}
```

Теперь мы можем написать фильтры для класса `Melon`:

```
List<Melon> watermelons = Filters.filter(  
    melons, (Melon m) -> "Watermelon".equalsIgnoreCase(m.getType()));
```

Мы также можем сделать то же самое для чисел:

```
List<Integer> numbers = Arrays.asList(1, 13, 15, 2, 67);  
List<Integer> smallThan10 = Filters  
    .filter(numbers, (Integer i) -> i < 10);
```

Отступите на шаг назад и посмотрите, где мы начинали и где находимся сейчас. Огромная разница, и все благодаря функциональным интерфейсам Java 8 и лямбда-выражениям. Обратите ли вы внимание на аннотацию `@FunctionalInterface` в интерфейсе `Predicate`? Так вот, это тот тип информативной аннотации, который используется для обозначения функционального интерфейса. Это полезно в случае возникновения ошибки, если помеченный интерфейс не является функциональным.

В концептуальном плане функциональный интерфейс имеет ровно один абстрактный метод. Кроме того, интерфейс `Predicate`, который мы определили, уже существует в Java 8 как интерфейс `java.util.function.Predicate`. Пакет `java.util.function` содержит более 40 таких интерфейсов. Следовательно, перед определением нового пакета настоятельно рекомендуется проверить содержимое этого пакета. В большинстве случаев с работой справятся шесть стандартных встроенных функциональных интерфейсов. Они перечислены ниже:

◆ <code>Predicate<T></code> ;	◆ <code>Function<T, R></code> ;
◆ <code>Consumer<T></code> ;	◆ <code>UnaryOperator<T></code> ;
◆ <code>Supplier<T></code> ;	◆ <code>BinaryOperator<T></code> .

Функциональные интерфейсы и лямбда-выражения составляют отличную команду. Лямбда-выражения поддерживают имплементацию абстрактного метода функционального интерфейса непосредственно внутри строки кода. В сущности, все выражение воспринимается как пример конкретной имплементации функционального интерфейса, как показано в следующем фрагменте коде:

```
Predicate<Melon> predicate = (Melon m)
    -> "Watermelon".equalsIgnoreCase(m.getType());
```

167. Лямбда-выражение в двух словах

Анализ лямбда-выражения позволяет выявить три основные части (рис. 8.5).



Рис. 8.5

Ниже приводится описание каждой части лямбда-выражения.

- ◆ С левой стороны от стрелки мы имеем параметры этого лямбда-выражения, которые используются в его теле. Это параметры метода `FilenameFilter.accept(File folder, String fileName)`.
- ◆ С правой стороны от стрелки мы имеем тело лямбда-выражения, в данном случае проверяющее, можно ли прочитать папку, в которой был найден файл, и заканчивается ли имя файла суффиксом `.pdf`.
- ◆ Стрелка — это просто разделитель параметров и тела лямбда-выражения.

Версия анонимного класса этого лямбда-выражения выглядит следующим образом:

```
FilenameFilter filter = new FilenameFilter() {
    @Override
    public boolean accept(File folder, String fileName) {
        return folder.canRead() && fileName.endsWith(".pdf");
    }
};
```

Теперь, если мы посмотрим на лямбда-выражение и его анонимную версию, то можем заключить, что лямбда-выражение — это сжатая анонимная функция, которая может передаваться в качестве аргумента методу или храниться в переменной. Мы можем заключить, что лямбда-выражение может быть описано четырьмя словами (рис. 8.6).

Лямбда-выражения поддерживают параметризацию поведения, и это большой плюс (вернитесь к предыдущей задаче, где дается подробное тому объяснение). Наконец, имейте в виду, что лямбда-выражения могут использоваться только в контексте функционального интерфейса.



Рис. 8.6

168. Имплементирование шаблона исполнения вокруг опорной операции

Шаблон исполнения вокруг опорной операции¹ пытается устранить *стереотипный* код, который окружает конкретные операции. Например, операции, специфичные для файла, должны быть окружены кодом открытия и закрытия файла.

Шаблон исполнения вокруг опорной операции полезен главным образом в сценариях, подразумевающих операции, которые выполняются в пределах срока жизни ресурса после его открытия и перед его закрытием. Например, допустим, что у нас есть `Scanner` и наша первая операция состоит в считывании значения типа `double` из файла:

```
try (Scanner scanner = new Scanner(
    Path.of("doubles.txt"), StandardCharsets.UTF_8)) {

    if (scanner.hasNextDouble()) {
        double value = scanner.nextDouble();
    }
}
```

Позже еще одна операция состоит в печати всех значений типа `double`:

```
try (Scanner scanner = new Scanner(
    Path.of("doubles.txt"), StandardCharsets.UTF_8)) {

    while (scanner.hasNextDouble()) {
        System.out.println(scanner.nextDouble());
    }
}
```

На рис. 8.7 выделен стереотипный код, окружающий эти две операции.

¹ Идиома исполнения вокруг опорной операции (execute around) освобождает пользователя от определенных действий, которые всегда должны выполняться до и/или после бизнес-метода. Хорошим ее примером является выделение и высвобождение ресурсов, в результате чего пользователю остается указать только то, что делать с ресурсом. См. <https://java-design-patterns.com/patterns/execute-around/>. — Прим. перев.

```
try (Scanner scanner = new Scanner(  
    Path.of("doubles.txt"), StandardCharsets.UTF_8)) {  
}
```

Рис. 8.7

Во избежание этого стереотипного кода шаблон исполнения вокруг опорной операции опирается на параметризацию поведения (более подробно описанную в разд. 166 *"Написание функциональных интерфейсов"* ранее в этой главе). Вот шаги, которые необходимо предпринять для того, чтобы этого достичь:

1. Первым шагом является определение функционального интерфейса, который соответствует сигнатуре `Scanner -> double` и способен выбрасывать исключение `IOException`:

```
@FunctionalInterface  
public interface ScannerDoubleFunction {  
    double readDouble(Scanner scanner) throws IOException;  
}
```

Объявление функционального интерфейса является лишь половиной решения.

2. Пока что мы можем только написать лямбда-выражение типа `Scanner -> double`, но нам нужен метод, который принимает ее и исполняет. Для этого рассмотрим следующий метод в служебном классе `Double`:

```
public static double read(ScannerDoubleFunction snf)  
    throws IOException {  
  
    try (Scanner scanner = new Scanner(  
        Path.of("doubles.txt"), StandardCharsets.UTF_8)) {  
  
        return snf.readDouble(scanner);  
    }  
}
```

Лямбда-выражение, которое передается методу `read()`, выполняется внутри тела этого метода. Когда мы передаем лямбда-выражение, то предоставляем имплементацию абстрактного метода `readDouble()` непосредственно внутри строки кода. Он (метод) воспринимается как экземпляр функционального интерфейса `ScannerDoubleFunction`, и поэтому мы можем вызвать метод `readDouble()` для получения желаемого результата.

3. Теперь мы можем просто передавать наши операции в качестве лямбда-выражений и повторно использовать метод `read()`. Например, наши операции могут быть обернуты в два статических метода, как показано здесь (этот практический прием необходим для получения чистого кода и избежания больших лямбда-выражений):

```
private static double getFirst(Scanner scanner) {  
    if (scanner.hasNextDouble()) {
```



```
        return scanner.nextDouble();
    }

    return Double.NaN;
}

private static double sumAll(Scanner scanner) {
    double sum = 0.0d;

    while (scanner.hasNextDouble()) {
        sum += scanner.nextDouble();
    }

    return sum;
}
```

4. Имея эти две операции в качестве примеров, мы можем написать и другие операции. Давайте передадим их в метод `read()`:

```
double singleDouble = Doubles.read((Scanner sc) -> getFirst(sc));
double sumAllDoubles = Doubles.read((Scanner sc) -> sumAll(sc));
```

Шаблон исполнения вокруг опорной операции весьма полезен в целях исключения *стереотипного кода*, специфичного для открытия и закрытия ресурсов (операций ввода-вывода).

169. Имплементирование шаблона фабрики

Шаблон фабрики, или фабричный шаблон, позволяет нам создавать несколько типов объектов, не выставляя процесс создания экземпляра наружу вызывающему коду. Благодаря этому, мы можем скрыть сложный и/или чувствительный процесс создания объектов и предоставить вызывающему коду интуитивно понятную и простую в использовании фабрику объектов.

В классической имплементации фабричный шаблон опирается на интернированную инструкцию `switch()`, как показано в следующем примере:

```
public static Fruit newInstance(Class<?> clazz) {
    switch (clazz.getSimpleName()) {
        case "Gac":
            return new Gac();
        case "Hemi":
            return new Hemi();
        case "Cantaloupe":
            return new Cantaloupe();
        default:
            throw new IllegalArgumentException(
                "Недопустимый аргумент clazz: " + clazz);
    }
}
```

Здесь `Gac`, `Hemi` и `Cantaloupe` имплементируют один и тот же интерфейс `Fruit` и имеют пустой конструктор. Если этот метод располагается в служебном классе с именем `MelonFactory`, то мы можем вызвать его следующим образом:

```
Gac gac = (Gac) MelonFactory.newInstance(Gac.class);
```

Однако функциональный стиль Java 8 позволяет нам обратиться к конструкторам, используя технический прием, именуемый *ссылками на методы*. Он означает, что мы можем определить поставщика `Supplier<Fruit>` как ссылающегося на пустой конструктор `Gac` следующим образом:

```
Supplier<Fruit> gac = Gac::new;
```

Как насчет `Hemi`, `Cantaloupe` и т. д.? Что ж, все просто — мы можем поместить их все в отображение `Map` (обратите внимание, что здесь не создается экземпляр типа `Melon`; это просто ленивые *ссылки на методы*):

```
private static final Map<String, Supplier<Fruit>> MELONS
    = Map.of("Gac", Gac::new, "Hemi", Hemi::new, "Cantaloupe", Cantaloupe::new);
```

Далее мы можем переписать метод `newInstance()` так, чтобы он использовал это отображение:

```
public static Fruit newInstance(Class<?> clazz) {
    Supplier<Fruit> supplier = MELONS.get(clazz.getSimpleName());

    if (supplier == null) {
        throw new IllegalArgumentException(
            "Недопустимый аргумент clazz: " + clazz);
    }

    return supplier.get();
}
```

Вызывающий код не нуждается в дальнейших изменениях:

```
Gac gac = (Gac) MelonFactory.newInstance(Gac.class);
```

Однако очевидно, что конструкторы не всегда являются пустыми. Например, следующий ниже класс `Melon` содержит один конструктор с тремя аргументами:

```
public class Melon implements Fruit {
    private final String type;
    private final int weight;
    private final String color;

    public Melon(String type, int weight, String color) {
        this.type = type;
        this.weight = weight;
        this.color = color;
    }
}
```

Создать экземпляр этого класса посредством пустого конструктора не получится. Но если мы определим функциональный интерфейс, который поддерживает три аргумента и возврат, мы снова окажемся на верном пути:

```
@FunctionalInterface
public interface TriFunction<T, U, V, R> {
    R apply(T t, U u, V v);
}
```

На этот раз следующая ниже инструкция попытается извлечь конструктор с тремя аргументами типа `String`, `Integer` и `String`:

```
private static final
    TriFunction<String, Integer, String, Melon> MELON = Melon::new;
```

Метод `newInstance()`, который был создан специально для класса `Melon`, имеет вид:

```
public static Fruit newInstance(String name, int weight, String color) {
    return MELON.apply(name, weight, name);
}
```

Экземпляр класса `Melon` можно создать следующим образом:

```
Melon melon = (Melon) MelonFactory.newInstance("Gac", 2000, "red");
```

Готово! Теперь у нас есть фабрика экземпляров класса `Melon` посредством функциональных интерфейсов.

170. Имплементирование шаблона стратегий

Классический шаблон стратегий является довольно простым. Он состоит из интерфейса, представляющего семейство алгоритмов (стратегий) и нескольких имплементаций этого интерфейса (каждая имплементация является стратегией).

Например, следующий интерфейс объединяет стратегии удаления символов из данной строки:

```
public interface RemoveStrategy {
    String execute(String s);
}
```

Сначала мы определяем стратегию удаления числовых значений из строки:

```
public class NumberRemover implements RemoveStrategy {
    @Override
    public String execute(String s) {
        return s.replaceAll("\\d", "");
    }
}
```

Затем задаем стратегию удаления пробелов из строки:

```
public class WhitespacesRemover implements RemoveStrategy {
    @Override
    public String execute(String s) {
        return s.replaceAll("\\s", "");
    }
}
```

Наконец, давайте определим служебный класс, который действует как точка входа для стратегий:

```
public final class Remover {  
    private Remover() {  
        throw new AssertionError("Не получается создать экземпляр");  
    }  
  
    public static String remove(String s, RemoveStrategy strategy) {  
        return strategy.execute(s);  
    }  
}
```

Эта имплементация шаблона стратегий является простой и классической. Если мы хотим удалить из строки числовые значения, то можем сделать это следующим образом:

```
String text = "This is a text from 20 April 2050";  
String noNr = Remover.remove(text, new NumberRemover());
```

Но действительно ли нам нужны классы `NumberRemover` и `WhitespacesRemover`? Необходимо ли писать подобные классы для дальнейших стратегий? Совершенно очевидно, что нет.

Вгляните на наш интерфейс еще раз:

```
@FunctionalInterface  
public interface RemoveStrategy {  
    String execute(String s);  
}
```

Мы только что добавили подсказку `@FunctionalInterface`, потому что интерфейс `RemoveStrategy` определяет один-единственный абстрактный метод, и поэтому он является функциональным интерфейсом.

Что мы можем использовать в контексте функционального интерфейса? Что ж, ответ очевиден — лямбда-выражения. Более того, в данном сценарии лямбда-выражение может удалить стереотипный код (в нашем случае классы, представляющие стратегии) и инкапсулировать стратегию в своем теле:

```
String noNr = Remover.remove(text, s -> s.replaceAll("\\d", ""));  
String noWs = Remover.remove(text, s -> s.replaceAll("\\s", ""));
```

Вот так выглядит шаблон стратегий посредством лямбда-выражений.

171. Имплементирование шаблона трафаретного метода

Трафаретный метод (Template Method) — это классический шаблон архитектурного дизайна от "банды четырех", который позволяет нам писать скелет алгоритма в методе и переносить некоторые шаги этого алгоритма в клиентские подклассы.

Например, приготовление пиццы включает три основных шага: раскатать тесто, добавить начинку и выпечь пиццу. В то время как первый и последний шаги можно считать одинаковыми (фиксированные шаги) для всех видов пиццы, второй шаг отличается для каждого вида пиццы (переменный шаг).

Если мы поместим это в исходный код посредством шаблона трафаретного метода, то получим что-то вроде следующего (метод `make()` представляет трафаретный метод и содержит фиксированные и переменные шаги в четко заданном порядке):

```
public abstract class PizzaMaker {  
    public void make(Pizza pizza) {  
        makeDough(pizza);  
        addTopIngredients(pizza);  
        bake(pizza);  
    }  
  
    private void makeDough(Pizza pizza) {  
        System.out.println("Приготовить тесто");  
    }  
  
    private void bake(Pizza pizza) {  
        System.out.println("Испечь пиццу");  
    }  
  
    public abstract void addTopIngredients(Pizza pizza);  
}
```

Фиксированные шаги имеют заданные по умолчанию имплементации, в то время как переменный шаг представлен абстрактным методом под названием `addTopIngredients()`. Этот метод имплементируется подклассами данного класса. Например, неаполитанская пицца будет абстрагирована следующим образом:

```
public class NeapolitanPizza extends PizzaMaker {  
    @Override  
    public void addTopIngredients(Pizza p) {  
        System.out.println("Добавить: свежую моцареллу, помидоры,  
        листья базилика, орегано и оливковое масло ");  
    }  
}
```

А процесс приготовления греческой пиццы будет выглядеть так:

```
public class GreekPizza extends PizzaMaker {  
    @Override  
    public void addTopIngredients(Pizza p) {  
        System.out.println("Добавить: соус и сыр");  
    }  
}
```

Таким образом, каждый вид пиццы требует нового класса, который переопределяет метод `addTopIngredients()`. В конце мы можем приготовить пиццу вот так:

```
Pizza nPizza = new Pizza();
PizzaMaker nMaker = new NeapolitanPizza();
nMaker.make(nPizza);
```

Недостатком такого подхода является стереотипный код и многословие. Однако мы можем устранить этот недостаток с помощью лямбда-выражений. Мы можем представить переменные шаги трафаретного метода в виде лямбда-выражений. В зависимости от ситуации мы должны выбрать правильные функциональные интерфейсы. В нашем случае мы можем опереться на потребителя `Consumer` следующим образом:

```
public class PizzaLambda {
    public void make(Pizza pizza, Consumer<Pizza> addTopIngredients) {
        makeDough(pizza);
        addTopIngredients.accept(pizza);
        bake(pizza);
    }

    private void makeDough(Pizza p) {
        System.out.println("Приготовить тесто");
    }

    private void bake(Pizza p) {
        System.out.println("Испечь пиццу");
    }
}
```

На этот раз нет необходимости определять подклассы (не нужно иметь `NeapolitanPizza`, `GreekPizza` или др.). Мы просто передаем переменный шаг через лямбда-выражение. Давайте приготовим сицилийскую пиццу:

```
Pizza sPizza = new Pizza();
new PizzaLambda().make(sPizza, (Pizza p)
    -> System.out.println("Добавить: кусочки помидора, лука,
        анчоусов и зелени "));
```

Готово! Стереотипный код больше не нужен. Вариант на основе лямбда-выражения принципиально улучшил решение задачи.

172. Имплементирование шаблона наблюдателя

Шаблон наблюдателя опирается на объект (так называемого **агента**), который автоматически уведомляет своих подписчиков (так называемых **наблюдателей**), когда происходят какие-то события.

Например, *агентом* может быть штаб противопожарной службы, а *наблюдателями* — местные пожарные части. Когда начинается пожар, штаб противопожарной

службы уведомляет все местные пожарные станции и отправляет им адрес, по которому возник пожар.

Каждый наблюдатель анализирует полученный адрес и в зависимости от различных критериев принимает решение о тушении пожара.

Все местные пожарные части сгруппированы посредством интерфейса под названием `FireObserver`. Этот интерфейс определяет один-единственный абстрактный метод, который вызывается штабом противопожарной службы (*субъектом*):

```
public interface FireObserver {  
    void fire(String address);  
}
```

Каждая местная пожарная часть (*наблюдатель*) имплементирует этот интерфейс и решает, тушить пожар или нет, в имплементации метода `fire()`. Здесь у нас есть три местные пожарные части (Брукхейвен, Винингс и Декейтер):

```
public class BrookhavenFireStation implements FireObserver {  
    @Override  
    public void fire(String address) {  
        if (address.contains("Брукхейвен")) {  
            System.out.println(  
                "Пожарная часть Брукхейвена отправится на этот пожар");  
        }  
    }  
}  
  
public class ViningsFireStation implements FireObserver {  
    // тот же код, что и выше, но для ViningsFireStation  
}  
  
public class DecaturFireStation implements FireObserver {  
    // тот же код, что и выше, но для DecaturFireStation  
}
```

Половина работы сделана! Теперь нам нужно зарегистрировать этих *наблюдателей*, чтобы они уведомлялись *агентом*. Другими словами, каждая местная пожарная часть должна быть зарегистрирована как *наблюдатель* в штабе противопожарной службы (*агент*). Для этого мы объявляем еще один интерфейс, который определяет *агентский контракт* для регистрации и уведомления своих наблюдателей:

```
public interface FireStationRegister {  
    void registerFireStation(FireObserver fo);  
    void notifyFireStations(String address);  
}
```

Наконец, мы можем написать штаб противопожарной службы (агента):

```
public class FireStation implements FireStationRegister {  
    private final List<FireObserver> fireObservers = new ArrayList<>();
```

```
@Override
public void registerFireStation(FireObserver fo) {
    if (fo != null) {
        fireObservers.add(fo);
    }
}

@Override
public void notifyFireStations(String address) {
    if (address != null) {
        for (FireObserver fireObserver: fireObservers) {
            fireObserver.fire(address);
        }
    }
}
```

Теперь давайте зарегистрируем три наши местные пожарные части (*наблюдателей*) в штабе противопожарной службы (*агенте*):

```
FireStation fireStation = new FireStation();
fireStation.registerFireStation(new BrookhavenFireStation());
fireStation.registerFireStation(new DecaturFireStation());
fireStation.registerFireStation(new ViningsFireStation());
```

Теперь при возникновении пожара штаб противопожарной службы уведомит все зарегистрированные местные пожарные части:

```
fireStation.notifyFireStations(
    "Пожарная тревога: Уэстхейвен в Винингс 5901 Саффекс Грин в Атланте");
```

Шаблон наблюдателя был там успешно имплементирован.

Это еще один классический случай *стереотипного кода*. Каждая местная пожарная часть нуждается в новом классе и имплементации метода `fire()`.

Однако нам снова поможет лямбда-выражение! Взгляните на интерфейс `FireObserver`. У него один абстрактный метод, поэтому он является функциональным интерфейсом:

```
@FunctionalInterface
public interface FireObserver {
    void fire(String address);
}
```

Указанный функциональный интерфейс является аргументом метода `Fire.registerFireStation()`. В данном контексте мы можем передавать этому методу лямбда-выражение вместо нового экземпляра местной пожарной части. Лямбда-выражение будет содержать поведение в своем теле; поэтому мы можем удалить

классы местных пожарных частей и опереться на лямбда-выражения следующим образом:

```
fireStation.registerFireStation((String address) -> {
    if (address.contains("Брукхейвен")) {
        System.out.println("Пожарная часть Брукхейвена отправится на этот пожар");
    }
});

fireStation.registerFireStation((String address) -> {
    if (address.contains("Винингс")) {
        System.out.println("Пожарная часть Винингса отправится на этот пожар");
    }
});

fireStation.registerFireStation((String address) -> {
    if (address.contains("Декейтер")) {
        System.out.println("Пожарная часть Декейтера отправится на этот пожар");
    }
});
```

Готово! Больше нет никакого стереотипного кода.

173. Имплементирование шаблона одалживания

В этой задаче мы поговорим об имплементировании шаблона одалживания. Допустим, что у нас есть файл, содержащий три числа (скажем, числа двойной точности), и каждое число является коэффициентом формулы. Например, числа x , y и z являются коэффициентами следующих двух формул: $x+y-z$ и $x-y*\sqrt{z}$. Таким же образом мы можем написать и другие формулы.

На данный момент у нас достаточно опыта, чтобы признать, что этот сценарий хорошо подходит для параметризации поведения. На этот раз мы не определяем собственный функциональный интерфейс, а используем встроенный функциональный интерфейс под названием `Function<T, R>`. Этот функциональный интерфейс представляет собой функцию, которая принимает один аргумент и выдает результат. Сигнатура его абстрактного метода имеет вид `R apply (T t)`.

Этот функциональный интерфейс становится аргументом статического метода, предназначенного для имплементации шаблона одалживания. Давайте поместим этот метод в класс `Formula`:

```
public class Formula {
    ...
    public static double compute(
        Function<Formula, Double> f) throws IOException {
        ...
    }
}
```

Обратите внимание, что метод `compute()` принимает лямбда-выражения типа `Formula -> Double`, когда он объявляется в классе `Formula`. Давайте раскроем весь исходный код метода `compute()`:

```
public static double compute(Function<Formula, Double> f) throws IOException {
    Formula formula = new Formula();
    double result = 0.0 d;

    try {
        result = f.apply(formula);
    } finally {
        formula.close();
    }

    return result;
}
```

Здесь следует выделить три момента. Во-первых, когда мы создаем новый экземпляр класса `Formula`, мы фактически открываем новый `Scanner` в нашем файле (взгляните на приватный конструктор этого класса):

```
public class Formula {
    private final Scanner scanner;
    private double result;

    private Formula() throws IOException {
        result = 0.0 d;

        scanner = new Scanner(
            Path.of("doubles.txt"), StandardCharsets.UTF_8);
    }
    ...
}
```

Во-вторых, когда мы выполняем лямбда-выражение, мы фактически вызываем цепочку методов экземпляра класса `Formula`, которые проводят вычисление (применяют формулу). Каждый из этих методов возвращает текущий экземпляр. Вызываемые экземплярные методы определяются в теле лямбда-выражения.

Нам нужны только вот эти вычисления, но можно добавить и другие:

```
public Formula add() {
    if (scanner.hasNextDouble()) {
        result += scanner.nextDouble();
    }

    return this;
}

public Formula minus() {
    if (scanner.hasNextDouble()) {
```

```
        result -= scanner.nextDouble();
    }

    return this;
}

public Formula multiplyWithSqrt() {
    if (scanner.hasNextDouble()) {
        result *= Math.sqrt(scanner.nextDouble());
    }

    return this;
}
```

Поскольку результатом вычисления (формула) является тип `double`, мы должны обеспечить терминальный метод, возвращающий финальный результат:

```
public double result() {
    return result;
}
```

Наконец, мы закрываем сканер и сбрасываем результат. Это происходит в приватном методе `close()`:

```
private void close() {
    try (scanner) {
        result = 0.0 d;
    }
}
```

Эти фрагменты включены в исходный код, прилагаемый к данной книге, под классом `Formula`.

Вспомните наши формулы: $x+y-z$ и $x-y*\sqrt{z}$. Первую можно написать следующим образом:

```
double xPlusYMinusZ = Formula.compute((sc)
    -> sc.add().add().minus().result());
```

Вторая формула может быть написана так:

```
double xMinusYMultiplySqrtZ = Formula.compute((sc)
    -> sc.add().minus().multiplyWithSqrt().result());
```

Обратите внимание, что мы можем сосредоточиться на наших формулах, и нам не нужно беспокоиться об открытии и закрытии файла. Кроме того, гибкий API позволяет нам образовывать любую формулу, и его очень легко обогатить большим числом операций.

174. Имплементирование шаблона декоратора

Шаблон декоратора предпочитает композицию наследованию, поэтому он является элегантной альтернативой подклассированию. При этом мы главным образом начинаем с базового объекта и добавляем дополнительные функции в динамическом режиме.

Например, мы можем использовать этот шаблон для украшения (декорирования) торта. Процесс декорирования не меняет сам торт — он просто добавляет немного орехов, взбитых сливок, фруктов и т. д.

Схема на рис. 8.8 иллюстрирует то, что мы будем имплементировать.

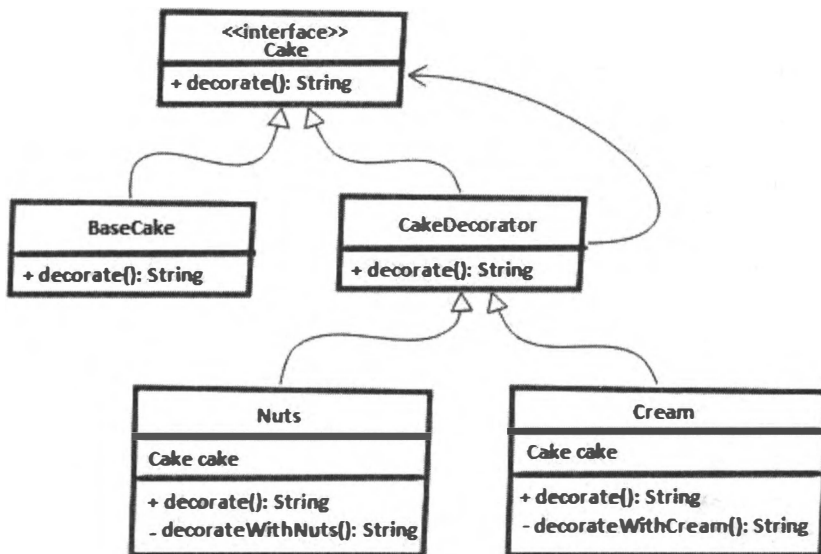


Рис 8.8

Сначала мы создаем интерфейс **Cake** (торт):

```
public interface Cake {
    String decorate();
}
```

Затем мы имплементируем этот интерфейс посредством **BaseCake**:

```
public class BaseCake implements Cake {
    @Override
    public String decorate() {
        return "Базовый торт ";
    }
}
```

После этого мы создаем абстрактный класс **CakeDecorator** для этого торта. Главная цель этого класса — вызывать метод **decorate()** для заданного торта:

```
public class CakeDecorator implements Cake {
    private final Cake cake;

    public CakeDecorator(Cake cake) {
        this.cake = cake;
    }
}
```

```
@Override
public String decorate() {
    return cake.decorate();
}
}
```

Затем мы сосредоточимся на написании наших декораторов.

Каждый декоратор расширяет класс `CakeDecorator` и изменяет метод `decorate()`, чтобы добавить соответствующее декорирование.

Например, декоратор орехов `Nuts` выглядит так:

```
public class Nuts extends CakeDecorator {
    public Nuts(Cake cake) {
        super(cake);
    }

    @Override
    public String decorate() {
        return super.decorate() + decorateWithNuts();
    }

    private String decorateWithNuts() {
        return "с орехами ";
    }
}
```

Для краткости мы пропустим декоратор `Cream` (крем). Однако довольно просто интуитивно понять, что этот декоратор в сущности такой же, как декоратор `Nuts`.

Итак, опять же, у нас есть какой-то *стереотипный код*.

Теперь мы можем создать торт, декорированный орехами и взбитыми сливками, следующим образом:

```
Cake cake = new Nuts(new Cream(new BaseCake()));
// Базовый торт со сливками с орехами
```

```
System.out.println(cake.decorate());
```

Таким образом, мы получили классическую имплементацию шаблона декоратора. Теперь давайте взглянем на имплементацию на основе лямбда-выражений, которая значительно сокращает этот код. В особенности это касается тех случаев, когда у нас имеется значительное число декораторов.

На этот раз мы выполним трансформацию интерфейса `Cake` в класс следующим образом:

```
public class Cake {
    private final String decorations;

    public Cake(String decorations) {
        this.decorations = decorations;
    }
}
```

```
public Cake decorate(String decoration) {  
    return new Cake(getDecorations() + decoration);  
}  
  
public String getDecorations() {  
    return decorations;  
}  
}
```

Кульминационным здесь является метод `decorate()`. Этот метод применяет заданную декорацию рядом с существующими декорациями и возвращает новый торт.

В качестве еще одного примера рассмотрим класс `java.awt.Color`, который имеет метод `brighter()`. Этот метод создает новый цвет `Color`, который является более яркой версией текущего `Color`. Схожим образом, метод `decorate()` создает новый торт `Cake`, который является более декорированной версией текущего торта.

Далее нет необходимости писать декораторы как отдельные классы. Мы будем опираться на лямбда-выражения для передачи декораторов в `CakeDecorator`:

```
public class CakeDecorator {  
    private Function<Cake, Cake> decorator;  
  
    public CakeDecorator(Function<Cake, Cake>... decorations) {  
        reduceDecorations(decorations);  
    }  
  
    public Cake decorate(Cake cake) {  
        return decorator.apply(cake);  
    }  
  
    private void reduceDecorations(  
        Function<Cake, Cake>... decorations) {  
        decorator = Stream.of(decorations)  
            .reduce(Function.identity(), Function::andThen);  
    }  
}
```

В общих чертах этот класс выполняет два действия.

- ◆ В конструкторе он вызывает метод `reduceDecorations()`. Этот метод будет связывать в цепочку массив передаваемых функций `Function` посредством методов `Stream.reduce()` и `Function.andThen()`. Результатом является одна-единственная функция `Function`, составленная из массива заданных функций `Function`.
- ◆ Когда метод `apply()` составленных функций `Function` вызывается из метода `decorate()`, он будет применять цепочку заданных функций одну за другой. Поскольку каждая функция в данном массиве является декоратором, составленная функция будет применять каждый декоратор по одному.

Давайте создадим торт, декорированный орехами и взбитыми сливками:

```
CakeDecorator nutsAndCream = new CakeDecorator(  
    (Cake c) -> c.decorate(" с орехами"),  
    (Cake c) -> c.decorate(" со взбитыми сливками"));  
  
Cake cake = nutsAndCream.decorate(new Cake("Базовый торт"));  
  
// Базовый торт с орехами и взбитыми сливками  
System.out.println(cake.getDecorations());
```

Готово! Попробуйте выполнить исходный код, прилагаемый к этой книге, чтобы проверить результаты.

175. Имплементирование шаблона каскадного строителя

Мы уже говорили об этом шаблоне в *разд. 51 "Написание немутуируемого класса посредством шаблона строителя" главы 2*. Рекомендуется вернуться к этой задаче, чтобы освежить в памяти принцип работы шаблона строителя.

Имея классического строителя в своем арсенале инструментов, предположим, что мы хотим написать класс для доставки посылок. В общих чертах мы хотим задать имя, фамилию, адрес получателя и содержимое посылки, а затем доставить посылку.

Мы можем выполнить это с помощью шаблона строителя и лямбда-выражений, как показано ниже:

```
public final class Delivery {  
    public Delivery firstname(String firstname) {  
        System.out.println(firstname);  
  
        return this;  
    }  
  
    // то же самое для фамилии, адреса и содержимого  
  
    public static void deliver(Consumer<Delivery> parcel) {  
        Delivery delivery = new Delivery();  
        parcel.accept(delivery);  
  
        System.out.println("\nГотово...");  
    }  
}
```

Для доставки посылки мы просто используем лямбда-выражение:

```
Delivery.deliver(d -> d.firstname("Mark")  
    .lastname("Kyilt")  
    .address("25 Street, New York")  
    .content("10 books"));
```

Очевидно, что применение лямбда-выражений облегчается аргументом

`Consumer<Delivery>`.

176. Имплементирование шаблона команд

Шаблон команд используется в сценариях, где команда обернута в объект. Этот объект может передаваться по кругу, не зная о самой команде или получателе команды.

Классическая имплементация этого шаблона состоит из нескольких классов. В нашем сценарии мы имеем следующее.

- ◆ Командный интерфейс отвечает за исполнение некоторого действия (в данном случае возможными действиями являются перемещение, копирование и удаление). Конкретными имплементациями этого интерфейса выступают `CopyCommand`, `MoveCommand` и `DeleteCommand`.
- ◆ Интерфейс `IODevice` определяет поддерживаемые действия (`move()`, `copy()` и `delete()`). Класс `HardDisk` является конкретной имплементацией интерфейса `IODevice` и представляет собой *получателя*.
- ◆ Класс `Sequence` является *инициатором* команд, и он знает, как исполнить заданную команду. Инициатор может действовать по-разному, но в данном случае мы просто записываем команды и выполняем их пакетно, когда вызывается метод `runSequence()`.

Шаблон команд можно представить схемой, приведенной на рис. 8.9.

Таким образом, класс `HardDisk` имплементирует действия, заданные в интерфейсе `IODevice`. В качестве *получателя* `HardDisk` отвечает за исполнение фактического действия при вызове метода `execute()` некоторой команды. Исходный код для `IODevice` выглядит следующим образом:

```
public interface IIODevice {  
    void copy();  
    void delete();  
    void move();  
}
```

Класс `HardDisk` представляет собой конкретную имплементацию интерфейса `IODevice`:

```
public class HardDisk implements IIODevice {  
    @Override  
    public void copy() {  
        System.out.println("Копирование...");  
    }  
  
    @Override  
    public void delete() {
```



```

        System.out.println("Удаление...");
    }

    @Override
    public void move() {
        System.out.println("Перемещение...");
    }
}

```

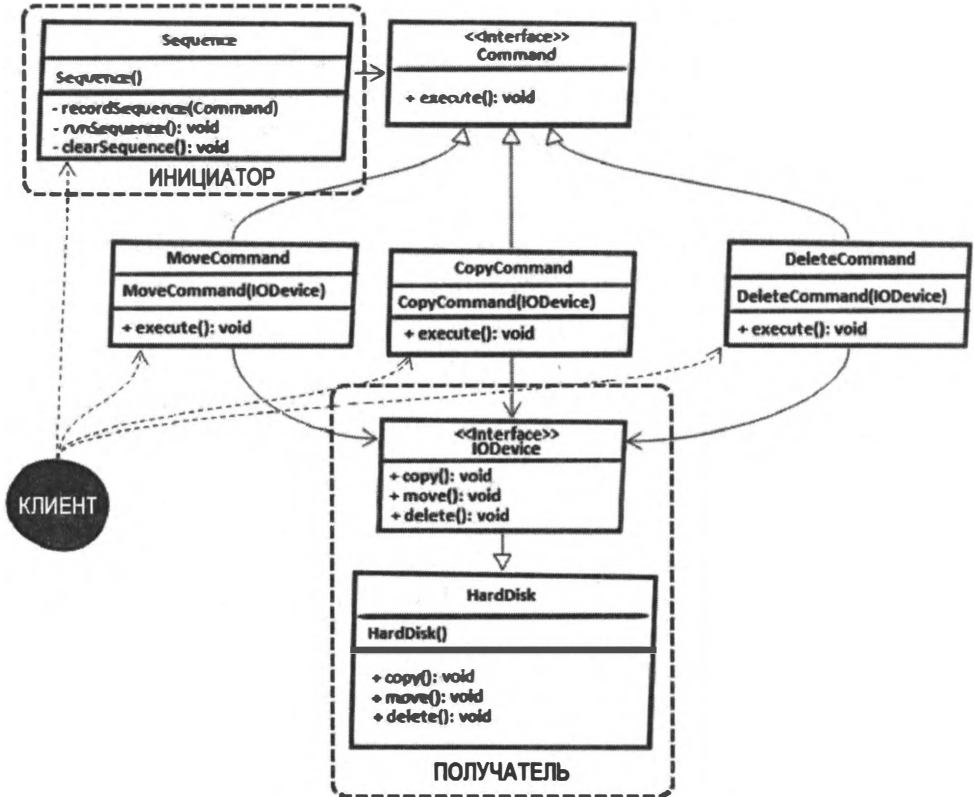


Рис. 8 9

Все конкретные классы команд имплементируют интерфейс Command:

```

public interface Command {
    public void execute();
}

public class DeleteCommand implements Command {
    private final IDevice action;

```

```
public DeleteCommand(IODevice action) {
    this.action = action;
}

@Override
public void execute() {
    action.delete()
}
}
```

Таким же образом мы могли бы имплементировать `CopyCommand` и `MoveCommand`, но пропустили их для краткости.

Далее класс `Sequence` действует как *инициирующий класс*. *Инициатор* знает, как исполнить заданную команду, но он не имеет никакого представления об имплементации этой команды (он знает только интерфейс команды). Здесь мы записываем команды в список и исполняем их пакетно при вызове метода `runSequence()`:

```
public class Sequence {
    private final List<Command> commands = new ArrayList<>();

    public void recordSequence(Command cmd) {
        commands.add(cmd);
    }

    public void runSequence() {
        commands.forEach(Command::execute);
    }

    public void clearSequence() {
        commands.clear();
    }
}
```

А теперь посмотрим, как это работает. Давайте исполним пакет действий на `HardDisk`:

```
HardDisk hd = new HardDisk();
Sequence sequence = new Sequence();
sequence.recordSequence(new CopyCommand(hd));
sequence.recordSequence(new DeleteCommand(hd));
sequence.recordSequence(new MoveCommand(hd));
sequence.recordSequence(new DeleteCommand(hd));
sequence.runSequence();
```

Очевидно, у нас здесь много *стереотипного кода*. Взгляните на классы команд. Действительно ли нам нужны все эти классы? Так вот, если мы понимаем, что интерфейс `Command` на самом деле является функциональным интерфейсом, мы можем

удалить его имплементации и обеспечить поведение посредством лямбда-выражений (классы команд — это просто блоки поведения, и поэтому они могут быть выражены посредством лямбда-выражения) следующим образом:

```
HardDisk hd = new HardDisk();
Sequence sequence = new Sequence();
sequence.recordSequence(hd::copy);
sequence.recordSequence(hd::delete);
sequence.recordSequence(hd::move);
sequence.recordSequence(hd::delete);
sequence.runSequence();
```

Резюме

Вот мы и подошли к концу очередной главы. Использование лямбда-выражений для сокращения или даже устранения стереотипного кода является как раз тем техническим приемом, который может использоваться и в других шаблонах архитектурного дизайна и сценариях. Знания, которые вы накопили к этому моменту, должны предоставить вам прочную основу для соответствующей адаптации исходного кода под ваши собственные случаи.

Вы можете скачать приложения из этой главы, чтобы увидеть результаты и получить дополнительные детали, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

9

Программирование в функциональном стиле — глубокое погружение

Эта глава содержит 22 задачи с привлечением функционального стиля программирования на языке Java. Здесь мы сосредоточимся на нескольких задачах, связанных с классическими операциями, которые встречаются в потоках (например, `filter` и `map`), и обсудим бесконечные потоки, `null`-безопасные потоки и методы по умолчанию. Этот полный список задач охватит группирование, разбиение и коллекторы, включая коллектор JDK 12 `teeing()` и написание собственного коллектора. В дополнение к этому будут рассмотрены функции `takeWhile()`, `dropWhile()`, композиция функций, предикатов и компараторов, тестирование и отладка лямбда-выражений и другие животрепещущие темы.

После того как вы прочтаете эту и предыдущую главы, вы будете готовы развернуть программирование в функциональном стиле в своих производственных приложениях. Рассмотренные далее задачи подготовят вас к широкому диапазону вариантов использования, включая крайние случаи и неприятные неожиданности.

Задачи

Используйте следующие задачи для проверки вашего умения программировать в функциональном стиле. Перед тем как обратиться к решениям и скачать примеры программ, настоятельно рекомендуется изучить каждую задачу.

177. **Тестирование функций высокого порядка.** Написать несколько модульных тестов для тестирования так называемых функций высокого порядка.

178. **Тестирование методов, использующих лямбда-выражения.** Написать несколько модульных тестов для тестирования методов, использующих лямбда-выражения.
179. **Отладка лямбда-выражений.** Предоставить техническое решение по отладке лямбда-выражений.
180. **Фильтрация непустых элементов потока.** Написать потоковый конвейер, который фильтрует непустые элементы потока.
181. **Бесконечные потоки, методы `takeWhile()` и `dropWhile()`.** Написать несколько фрагментов кода, которые работают с бесконечными потоками. Кроме того, написать несколько примеров работы с API методов `takeWhile()` и `dropWhile()`.
182. **Отображение элементов потока в новый поток.** Написать несколько примеров отображения элементов потока в новый поток посредством методов `map()` и `flatMap()`.
183. **Отыскание элементов в потоке.** Написать программу для отыскания разных элементов в потоке.
184. **Сопоставление элементов в потоке.** Написать программу для сопоставления разных элементов в потоке.
185. **Сумма, максимум и минимум в потоке.** Написать программу для вычисления суммы, максимума и минимума заданного потока посредством примитивных специализаций потока и метода `Stream.reduce()`.
186. **Сбор результатов потока.** Написать несколько фрагментов кода для сбора результатов потока в качестве списка `List`, отображения `Map` и множества `Set`.
187. **Соединение результатов потока.** Написать несколько фрагментов кода для соединения результатов потока в строковое значение.
188. **Подытоживающие коллекторы.** Написать несколько фрагментов кода, чтобы показать использование подытоживающих коллекторов.
189. **Группирование.** Написать несколько фрагментов кода для работы с коллекторами `groupingBy()`.
190. **Разбиение.** Написать несколько фрагментов кода для работы с коллекторами `partitioningBy()`.
191. **Фильтрующий, сглаживающий и отображающий коллекторы.** Написать несколько фрагментов кода для иллюстрации использования фильтрующего, сглаживающего и отображающего коллекторов.
192. **Сочленение.** Написать несколько примеров, которые объединяют результаты двух коллекторов (`JDK 12` и `Collectors.teeing()`).
193. **Написание собственного коллектора.** Написать программу, представляющую собственный специализированный коллектор.
194. **Ссылка на метод.** Написать пример ссылки на метод.

195. **Параллельная обработка потоков.** Провести краткий обзор параллельной обработки потоков. Привести хотя бы по одному примеру работы методов `parallelStream()`, `parallel()` и `splitterator()`.
196. **null-безопасные потоки.** Написать программу, которая возвращает null-безопасный поток из элемента или коллекции элементов.
197. **Композиция функций, предикатов и компараторов.** Написать несколько примеров для композиции функций, предикатов и компараторов.
198. **Методы по умолчанию.** Написать интерфейс, содержащий метод по умолчанию.

Решения

В следующих разделах описаны решения перечисленных выше задач. Помните, что обычно не существует одного единственно правильного варианта решения той или иной задачи. Кроме того, имейте в виду, что приведенные здесь объяснения включают только самые интересные и важные детали, необходимые для решения. Вы можете скачать примеры решений, чтобы увидеть дополнительные детали и поэкспериментировать с программами, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

177. Тестирование функций высокого порядка

Термин *"функция высокого порядка"* используется в качестве характеристики функции, которая возвращает функцию или берет функцию в качестве параметра.

Исходя из этого утверждения, тестирование функции высокого порядка в контексте лямбда-выражений должно охватывать два главных случая:

- ◆ тестирование метода, принимающего лямбда-выражение в качестве параметра;
- ◆ тестирование метода, возвращающего функциональный интерфейс.

Мы узнаем об этих двух проверках в следующих далее разделах.

Тестирование метода, принимающего лямбда-выражение в качестве параметра

Протестировать метод, принимающий лямбда-выражение в качестве параметра, можно путем передачи в этот метод разных лямбда-выражений. Например, допустим, что у нас есть следующий функциональный интерфейс:

```
@FunctionalInterface
public interface Replacer<String> {
    String replace(String s);
}
```

Кроме того, допустим, что у нас есть метод, который берет лямбда-выражения типа `String -> String`, как показано ниже:

```
public static List<String> replace(List<String> list, Replacer<String> r) {
    List<String> result = new ArrayList<>();
    for (String s: list) {
        result.add(r.replace(s));
    }

    return result;
}
```

Теперь давайте напишем модульный JUnit-тест для этого метода с использованием двух лямбда-выражений:

```
@Test
public void testReplacer() throws Exception {
    List<String> names = Arrays.asList(
        "Ann a 15", "Mir el 28", "D oru 33");

    List<String> resultWs = replace(
        names, (String s) -> s.replaceAll("\\s", ""));
    List<String> resultNr = replace(
        names, (String s) -> s.replaceAll("\\d", ""));

    assertEquals(Arrays.asList(
        "Anna15", "Mirel28", "Doru33"), resultWs);
    assertEquals(Arrays.asList(
        "Ann a ", "Mir el ", "D oru "), resultNr);
}
```

Тестирование метода, возвращающего функциональный интерфейс

Тестирование метода, возвращающего функциональный интерфейс, можно интерпретировать как тестирование поведения этого функционального интерфейса. Рассмотрим следующий метод:

```
public static Function<String, String> reduceStrings(
    Function<String, String> ...functions) {

    Function<String, String> function = Stream.of(functions)
        .reduce(Function.identity(), Function::andThen);

    return function;
}
```

Теперь мы можем протестировать поведение возвращаемой функции `Function<String, String>` следующим образом:

```
@Test
public void testReduceStrings() throws Exception {
    Function<String, String> f1 = (String s) -> s.toUpperCase();

    Function<String, String> f2 = (String s) -> s.concat(" DONE");

    Function<String, String> f = reduceStrings(f1, f2);

    assertEquals("TEST DONE", f.apply("test"));
}
```

178. Тестирование методов, использующих лямбда-выражения

Давайте начнем с тестирования лямбда-выражения, которое не завернуто в метод. Например, следующее лямбда-выражение ассоциировано с полем (в целях повторного использования) и мы хотим протестировать логику этого лямбда-выражения:

```
public static final Function<String, String> firstAndLastChar
    = (String s) -> String.valueOf(s.charAt(0))
        + String.valueOf(s.charAt(s.length() - 1));
```

Примем во внимание, что лямбда-выражение генерирует экземпляр функционального интерфейса; тогда мы можем протестировать поведение этого экземпляра следующим образом:

```
@Test
public void testFirstAndLastChar() throws Exception {
    String text = "Lambda";
    String result = firstAndLastChar.apply(text);
    assertEquals("La", result);
}
```



Еще один вариант решения состоит в том, чтобы обернуть лямбда-выражение в вызов метода и написать модульные тесты для вызова метода.

Часто лямбда-выражения используются внутри методов. В большинстве случаев совершенно приемлемо тестировать метод, содержащий лямбда-выражение, но есть случаи, когда мы хотим проверить само лямбда-выражение. Решение этой задачи состоит из трех главных шагов:

1. Извлечь лямбда-выражение в статический метод.
2. Заменить лямбда-выражение *ссылкой на метод*.
3. Протестировать этот статический метод.

Для примера рассмотрим следующий метод:

```
public List<String> rndStringFromStrings(List<String> strs) {
    return strs.stream()
        .map(str -> {
            Random rnd = new Random();
            int nr = rnd.nextInt(str.length());
            String ch = String.valueOf(str.charAt(nr));

            return ch;
        })
        .collect(Collectors.toList());
}
```

Наша цель — проверить лямбда-выражение из этого метода:

```
str -> {
    Random rnd = new Random();
    int nr = rnd.nextInt(str.length());
    String ch = String.valueOf(str.charAt(nr));

    return ch;
}
```

Итак, давайте применим упомянутые выше три шага.

1. Извлечем это лямбда-выражение в статический метод:

```
public static String extractCharacter(String str) {
    Random rnd = new Random();
    int nr = rnd.nextInt(str.length());
    String chAsStr = String.valueOf(str.charAt(nr));

    return chAsStr;
}
```

2. Заменяем лямбда-выражение соответствующей ссылкой на метод:

```
public List<String> rndStringFromStrings(List<String> strs) {
    return strs.stream()
        .map(StringOperations::extractCharacter)
        .collect(Collectors.toList());
}
```

3. Проверим статический метод (т. е. лямбда-выражение):

```
@Test
public void testRndStringFromStrings() throws Exception {
    String str1 = "Some";
    String str2 = "random";
    String str3 = "text";
}
```

```
String result1 = extractCharacter(str1);
String result2 = extractCharacter(str2);
String result3 = extractCharacter(str3);

assertEquals(result1.length(), 1);
assertEquals(result2.length(), 1);
assertEquals(result3.length(), 1);
assertThat(str1, containsString(result1));
assertThat(str2, containsString(result2));
assertThat(str3, containsString(result3));
}
```



Рекомендуется избегать лямбда-выражений, содержащих более одной строки кода. Таким образом, если следовать предыдущему техническому решению, то лямбда-выражения становятся легко проверяемыми.

179. Отладка лямбда-выражений

В том, что касается отладки лямбда-выражений, существуют по крайней мере три решения:

- ◆ осмотр трассы стека;
- ◆ журналирование/протоколирование;
- ◆ опора на поддержку со стороны IDE (например, NetBeans, Eclipse и IntelliJ IDEA поддерживают отладку лямбда-выражений из *коробки* либо предусматривают для них плагины).

Давайте сосредоточимся на первых двух, т. к. использование IDE — очень большая и специфическая тема, которая не входит в рамки настоящей книги.

Инспектирование стековой трассы сбоя, произошедшего внутри лямбда-конвейера или потокового конвейера, может стать довольно загадочным. Рассмотрим следующий ниже фрагмент кода:

```
List<String> names = Arrays.asList("anna", "bob", null, "mary");
```

```
names.stream()
    .map(s -> s.toUpperCase())
    .collect(Collectors.toList());
```

Поскольку третий элемент из этого списка равен `null`, мы получим исключение `NullPointerException`, и вся последовательность вызовов, определяющая потоковый конвейер, будет выставлена, как показано на рис. 9.1.

Выделенная строка говорит о том, что это исключение `NullPointerException` произошло внутри лямбда-выражения с именем `lambda$main$5`. Это имя было составлено компилятором, т. к. лямбда-выражения не имеют имен. Более того, мы не знаем, какой элемент был равен `null`.

```
Exception in thread "main" java.lang.NullPointerException
    at modern.challenge.Main.lambda$main$5(Main.java:28)
    at java.base/java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline.java:195)
    at java.base/java.util.Spliterators$ArraySpliterator.forEachRemaining(Spliterators.java:948)
    at java.base/java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:484)
    at java.base/java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:474)
    at java.base/java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:913)
    at java.base/java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)
    at java.base/java.util.stream.ReferencePipeline.collect(ReferencePipeline.java:578)
    at modern.challenge.Main.main(Main.java:29)
```

Рис. 9 1

Таким образом, мы можем заключить, что стековая трасса, которая сообщает о сбое внутри лямбда-конвейера или потокового конвейера, не очень понятна в интуитивном плане.

В качестве альтернативы мы можем попробовать протоколировать результат в журнал операций, что поможет нам отладить конвейер операций в потоке. Это можно сделать с помощью метода `forEach()`:

```
List<String> list = List.of("anna", "bob",
    "christian", "carmen", "rick", "carla");

list.stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

В результате мы получим следующее:

```
CARLA
CARMEN
CHRISTIAN
```

В некоторых случаях этот технический прием бывает полезным. Разумеется, мы должны иметь в виду, что операция `forEach()` является терминальной и поэтому поток будет потреблен полностью. Поскольку поток может потребляться только один раз, это может стать проблемой.

Более того, если мы добавим значение `null` в список, то результат снова станет запутанным.

Более качественная альтернатива состоит в том, чтобы опереться на метод `peek()`. Он представляет собой промежуточную операцию, которая выполняет определенное действие над текущим элементом и перенаправляет элемент к следующей операции в конвейере. На рис. 9.2 показан принцип работы метода `peek()`.

Давайте посмотрим на его работу в кодовой форме:

```
System.out.println("После:");

names.stream()
    .peek(p -> System.out.println("\tstream(): " + p))
    .filter(s -> s.startsWith("c"))
```

```
.peek(p -> System.out.println("\tfilter(): " + p))
.map(String::toUpperCase)
.peek(p -> System.out.println("\tmap(): " + p))
.sorted()
.peek(p -> System.out.println("\tsorted(): " + p))
.collect(Collectors.toList());
```

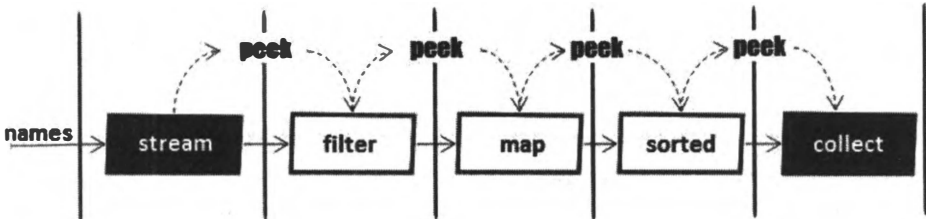


Рис. 9.2

На рис. 9.3 приведен пример результата, который мы можем получить.

```
After:
    stream(): anna
    stream(): bob
    stream(): christian
    filter(): christian
    map(): CHRISTIAN
    stream(): carmen
    filter(): carmen
    map(): CARMEN
    stream(): rick
    stream(): carla
    filter(): carla
    map(): CARLA
    sorted(): CARLA
    sorted(): CARMEN
    sorted(): CHRISTIAN
```

Рис. 9.3

Теперь давайте намеренно добавим значение `null` в список и выполним его снова:

```
List<String> names = Arrays.asList("anna", "bob",
    "christian", null, "carmen", "rick", "carla");
```

Результат, приведенный на рис. 9.4, был получен после добавления значения `null` в список.

На этот раз мы видим, что значение `null` возникло после применения метода `stream()`. Поскольку метод `stream()` является первой операцией, мы можем легко вычислить, что ошибка находится в содержимом списка.

После:

```
stream(): anna
stream(): bob
stream(): christian
filter(): christian
map(): CHRISTIAN
stream(): null
Exception in thread "main" java.lang.NullPointerException
    at modern.challenge.Main.lambda$main$1 (Main.java:16)
    ...
```

Рис. 9.4

180. Фильтрация непустых элементов потока

В разд. 166 "Написание функциональных интерфейсов" главы 8 мы определили метод filter(), основанный на функциональном интерфейсе с именем Predicate. API потоков Java уже имеет такой метод, и функциональный интерфейс называется java.util.function.Predicate.

Допустим, что у нас есть следующий список целых чисел:

```
List<Integer> ints = Arrays.asList(1, 2, -4, 0, 2, 0, -1, 14, 0, -1);
```

Потоковая передача этого списка и извлечение только не-null элементов может быть выполнена следующим образом:

```
List<Integer> result = ints.stream()
    .filter(i -> i != 0)
    .collect(Collectors.toList());
```

Результирующий список будет содержать следующие элементы: 1, 2, -4, 2, -1, 14, -1.

Схема на рис. 9.5 демонстрирует принцип работы метода filter() внутри.

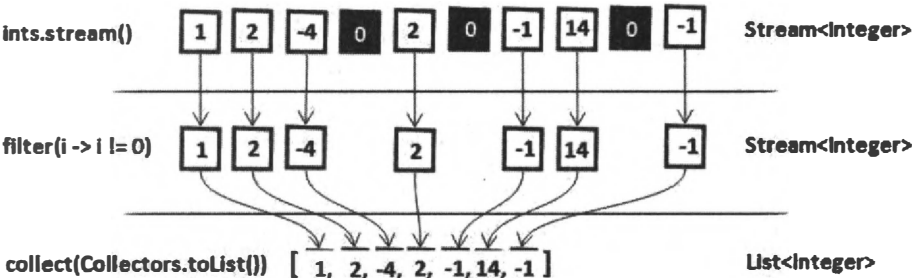


Рис. 9.5

Обратите внимание, что для нескольких часто встречающихся операций API потоков Java уже предусматривает готовые промежуточные операции. Следовательно, нет необходимости предоставлять Predicate. Некоторые из этих операций перечислены ниже:

- ◆ `distinct()` — удаляет повторы из потока;

- ◆ `skip(n)` — отбрасывает первые n элементов;
- ◆ `limit(s)` — усекает поток, делая его не длиннее s ;
- ◆ `sorted()` — сортирует поток в соответствии с естественным порядком;
- ◆ `sorted(Comparator<? super T> comparator)` — сортирует поток в соответствии с данным компаратором.

Давайте добавим эти операции и `filter()` в пример. Мы будем фильтровать нули и повторы, пропускать значение 1, усекать оставшийся поток до двух элементов и сортировать их в естественном порядке:

```
List<Integer> result = ints.stream()
    .filter(i -> i != 0)
    .distinct()
    .skip(1)
    .limit(2)
    .sorted()
    .collect(Collectors.toList());
```

Результирующий список будет содержать следующие два элемента: `-4` и `2`.

Схема на рис. 9.6 показывает принцип работы потокового конвейера внутри.

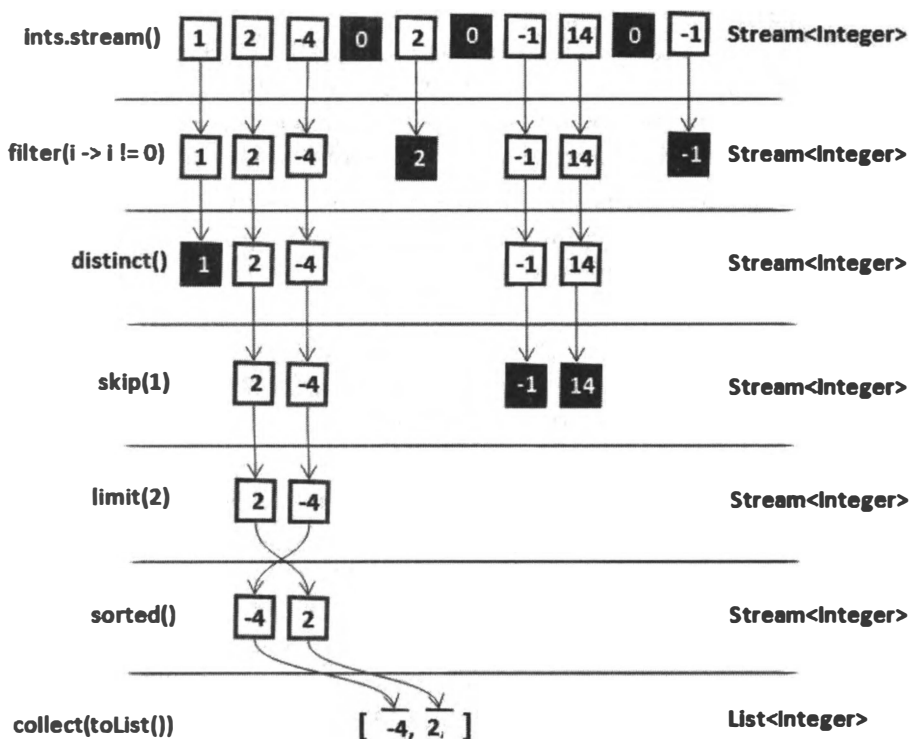


Рис. 9.6

Когда операция `filter()` требует сложное/составное или длительное состояние, то рекомендуется извлекать его во вспомогательный статический метод и опираться на *ссылку на метод*. Поэтому избегайте чего-то подобного:

```
List<Integer> result = ints.stream()
    .filter(value -> value > 0 && value < 10 && value % 2 == 0)
    .collect(Collectors.toList());
```

Вам следует предпочитать что-то вроде этого (`Numbers` — это класс, содержащий вспомогательный метод):

```
List<Integer> result = ints.stream()
    .filter(Numbers::evenBetween0And10)
    .collect(Collectors.toList());
```

```
private static boolean evenBetween0And10(int value) {
    return value > 0 && value < 10 && value % 2 == 0;
}
```

181. Бесконечные потоки, *takeWhile()* и *dropWhile()*

В первой части этой задачи мы поговорим о бесконечных потоках. Во второй части коснемся API методов `takeWhile()` и `dropWhile()`.

Бесконечный поток — это поток, который создает данные бесконечно. Поскольку потоки являются ленивыми, они могут быть бесконечными. Точнее, создание бесконечного потока выполняется как промежуточная операция, и поэтому никакие данные не создаются, пока не будет выполнена терминальная операция конвейера.

Например, следующий ниже фрагмент кода теоретически будет выполняться вечно. Это поведение активируется терминальной операцией `forEach()` и вызвано отсутствующим ограничением или лимитом:

```
Stream.iterate(1, i -> i + 1)
    .forEach(System.out::println);
```

API потоков Java позволяет нам создавать и управлять бесконечным потоком несколькими способами, как вы вскоре увидите.

Кроме того, поток `Stream` может быть *упорядоченным* или *неупорядоченным* в зависимости от заданного *порядка появления*. Наличие либо отсутствие у *потока порядка* зависит от источника данных и промежуточных операций. Например, поток `Stream`, источником которого является список `List`, является упорядоченным, поскольку список имеет внутреннее упорядочение. Поток, источником которого является множество `Set`, не является упорядоченным, поскольку множество не гарантирует порядка. Некоторые промежуточные операции (например, `sorted()`) могут накладывать порядок на неупорядоченный поток, в то время как некоторые терминальные операции (например, `forEach()`) могут игнорировать порядок появления.



Обычно упорядочивание влияет на производительность последовательных потоков незначительно, но в зависимости от применяемых операций производительность параллельных потоков может находиться под существенным воздействием упорядоченного потока.

Не путайте метод `Collection.stream().forEach()` с методом `Collection.forEach()`. отличие от метода `Collection.forEach()`, который может поддерживать порядок, опираясь на итератор коллекции (если таковой имеется), порядок метода `Collection.stream().forEach()` не определен. Например, неоднократный обход списка посредством метода `list.forEach()` обрабатывает элементы в порядке вставки, в то время как метод `list.parallelStream().forEach()` выдает разные результаты при каждом прогоне. В качестве общего правила, если поток не нужен, то следует выполнять обход коллекции посредством метода `Collection.forEach()`.

Мы можем преобразовать упорядоченный поток в неупорядоченный поток посредством метода `BaseStream.unordered()`, как показано в следующем примере:

```
List<Integer> list = Arrays.asList(1, 4, 20, 15, 2, 17, 5, 22, 31, 16);
```

```
Stream<Integer> unorderedStream = list.stream().unordered();
```

Бесконечный последовательный упорядоченный поток

Бесконечный последовательный упорядоченный поток можно получить посредством метода `Stream.iterate(T seed, UnaryOperator<T> f)`. Результирующий поток начинается с указанного начального элемента и продолжается путем применения функции `f` к предыдущему элементу (например, элемент `n` равен `f(n-1)`).

Поток целых чисел вида `1, 2, 3, ..., n` можно создать следующим образом:

```
Stream<Integer> infStream = Stream.iterate(1, i -> i + 1);
```

Далее мы можем использовать этот поток для различных целей. Например, давайте используем его для получения списка из первых 10 четных целых чисел:

```
List<Integer> result = infStream
    .filter(i -> i % 2 == 0)
    .limit(10)
    .collect(Collectors.toList());
```

Содержимое списка будет выглядеть следующим образом (обратите внимание, что бесконечный поток создаст элементы `1, 2, 3, ..., 20`, но только следующие ниже элементы совпадают с нашим фильтром до тех пор, пока не будет достигнут предел в 10 элементов):

```
2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```



Обратите внимание на наличие промежуточной операции `limit()`. Ее наличие является обязательным; в противном случае исходный код будет выполняться бесконечно. Мы должны явно отбросить поток; другими словами, мы должны явно указать, сколько совпадающих с нашим фильтром элементов должны быть собраны в финальном списке. Как только предел будет достигнут, бесконечный поток отбрасывается.

Но допустим, что нам не нужен список первых 10 четных целых чисел, а на самом деле нам требуется список четных целых чисел до 10 (или до любого другого предела). Начиная с JDK 9, мы можем сформировать это поведение с помощью новой разновидности метода `Stream.iterate()`. Эта разновидность позволяет нам встраивать предикат `hasNext` непосредственно в объявление потока (`iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)`). Поток terminates, как только предикат `hasNext` возвращает `false`:

```
Stream<Integer> infStream = Stream.iterate(
    1, i -> i <= 10, i -> i + 1);
```

На этот раз мы можем удалить промежуточную операцию `limit()`, т. к. наш предикат `hasNext` накладывает ограничение в 10 элементов:

```
List<Integer> result = infStream
    .filter(i -> i % 2 == 0)
    .collect(Collectors.toList());
```

Результирующий список выглядит следующим образом (в соответствии с нашим предикатом `hasNext` бесконечный поток создает элементы 1, 2, 3, ..., 10, но только следующие ниже пять элементов совпадают с нашим потоковым фильтром):

2, 4, 6, 8, 10

Конечно, мы можем объединить эту разновидность метода `Stream.iterate()` и метод `limit()` для формирования более сложных сценариев. Например, представленный далее поток будет создавать новый элемент до тех пор, пока *следующий предикат не будет равен* `i -> i <= 10`. Поскольку мы используем случайные значения, момент, когда предикат `hasNext` возвратит `false`, является недетерминированным:

```
Stream<Integer> infStream = Stream.iterate(
    1, i -> i <= 10, i -> i + i % 2 == 0
        ? new Random().nextInt(20) : -1 * new Random().nextInt(10));
```

Один из возможных результатов для этого потока выглядит так:

1, -5, -4, -7, -4, -2, -8, -8, ..., 3, 0, 4, -7, -6, 10, ...

Следующий конвейер будет собирать максимум 25 чисел, которые были созданы посредством `infStream`:

```
List<Integer> result = infStream
    .limit(25)
    .collect(Collectors.toList());
```

Теперь бесконечный поток можно отбросить из двух мест. Если предикат `hasNext` возвращает `false` до того, как мы соберем 25 элементов, мы останемся с элементами, собранными на тот момент (менее 25). Если предикат `hasNext` не возвращает `false` до того, как мы соберем 25 элементов, операция `limit()` отбросит остальную часть потока.

Нелимитированный поток псевдослучайных значений

Если мы хотим создавать нелимитированные потоки псевдослучайных значений, то можем опираться на методы класса `Random`, такие как `ints()`, `longs()` и `Double()`. Например, нелимитированный поток псевдослучайных целочисленных значений может быть объявлен следующим образом (генерируемые целые числа будут находиться в интервале `[1; 100]`):

```
IntStream rndInfStream = new Random().ints(1, 100);
```

Попытка получить список из 10 четных псевдослучайных целых значений может опираться на следующий поток:

```
List<Integer> result = rndInfStream
    .filter(i -> i % 2 == 0)
    .limit(10)
    .boxed()
    .collect(Collectors.toList());
```

Один из возможных результатов выглядит так:

```
8, 24, 82, 42, 90, 18, 26, 96, 86, 86
```

На этот раз трудно сказать, сколько чисел было фактически сгенерировано до тех пор, пока вышеупомянутый список не был собран.

Еще одной разновидностью метода `ints()` является метод `ints(long streamSize, int randomNumberOrigin, int randomNumberBound)`. Первый аргумент позволяет нам задавать число генерируемых псевдослучайных значений. Например, следующий ниже поток будет генерировать ровно 10 значений в интервале `[1; 100]`:

```
IntStream rndInfStream = new Random().ints(10, 1, 100);
```

Мы можем получить четные значения из этих 10 чисел следующим образом:

```
List<Integer> result = rndInfStream
    .filter(i -> i % 2 == 0)
    .boxed()
    .collect(Collectors.toList());
```

Один из возможных результатов выглядит так:

```
80, 28, 60, 54
```

Мы можем использовать этот пример в качестве основы для генерирования случайных строковых значений фиксированной длины:

```
IntStream rndInfStream = new Random().ints(20, 48, 126);
String result = rndInfStream
    .mapToObj(n -> String.valueOf((char) n))
    .collect(Collectors.joining());
```

Один из возможных результатов выглядит следующим образом:

```
AIW?Flob13KPKMI tqY8>
```



Метод `Stream.ints()` идет в комплекте с еще двумя разновидностями, одна из которых не берет никаких аргументов (нелимитированный поток целых чисел), и еще одной разновидностью, которая берет один аргумент, представляющий число значений, которые должны быть сгенерированы, т. е. `ints(long streamSize)`.

Бесконечный последовательный неупорядоченный поток

В создании бесконечного последовательного неупорядоченного потока мы можем опереться на метод `Stream.generate(Supplier<? extends T> s)`. В этом случае каждый элемент генерируется предоставленным поставщиком `Supplier`. Это подходит для генерирования постоянных потоков, потоков случайных элементов и т. д.

Например, допустим, что у нас есть простой помощник, который генерирует пароли из восьми символов:

```
private static String randomPassword() {
    String chars = "abcd0123!@#$";

    return new SecureRandom().ints(8, 0, chars.length())
        .mapToObj(i -> String.valueOf(chars.charAt(i)))
        .collect(Collectors.joining());
}
```

Далее мы хотим задать бесконечный последовательный неупорядоченный поток, который возвращает случайные пароли (`Main` — это класс, содержащий указанного выше помощника):

```
Supplier<String> passwordSupplier = Main::randomPassword;
Stream<String> passwordStream = Stream.generate(passwordSupplier);
```

На данный момент `passwordStream` может создавать пароли без ограничения. Но давайте создадим 10 таких паролей:

```
List<String> result = passwordStream
    .limit(10)
    .collect(Collectors.toList());
```

Один из возможных результатов выглядит следующим образом:

```
213c1blc, 2badc$21, d33321d$, @a0dc323, 3!1aa!dc, 0a3##@3!, $!b2#1d@,
0@0#dd$#, cb$12d2@, d2@ccc@d
```

Брать до тех пор, пока предикат возвращает *true*

Один из наиболее полезных методов, который был добавлен в класс `Stream`, начиная с JDK 9, был метод `takeWhile(Predicate<? super T> predicate)`. Этот метод идет в комплекте с двумя разными поведением.

- ♦ Если поток упорядочен, то этот метод возвращает поток, состоящий из *наибольшего префикса элементов*, взятых из этого потока, которые совпадают с заданным предикатом.

- ◆ Если поток не упорядочен и некоторые (но не все) элементы этого потока совпадают с заданным предикатом, то поведение этой операции не детерминировано; она свободна взять любое подмножество совпадающих элементов (которое включает пустое множество).

В случае упорядоченного потока наибольший префикс элементов — это непрерывная последовательность элементов потока, совпадающих с заданным предикатом.



Обратите внимание, что метод `takeWhile()` отбрасывает оставшийся поток, как только заданный предикат возвращает `false`.

Например, список из 10 целых чисел можно получить следующим образом:

```
List<Integer> result = IntStream
    .iterate(1, i -> i + 1)
    .takeWhile(i -> i <= 10)
    .boxed()
    .collect(Collectors.toList());
```

В результате мы получим:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

В качестве альтернативы мы можем получить список случайных четных целых чисел, ограниченный ситуацией, когда первое сгенерированное значение не будет меньше 50:

```
List<Integer> result = new Random().ints(1, 100)
    .filter(i -> i % 2 == 0)
    .takeWhile(i -> i >= 50)
    .boxed()
    .collect(Collectors.toList());
```

Мы даже можем соединить предикаты в методе `takeWhile()`:

```
List<Integer> result = new Random().ints(1, 100)
    .takeWhile(i -> i % 2 == 0 && i >= 50)
    .boxed()
    .collect(Collectors.toList());
```

Один из возможных результатов может быть получен следующим образом (он также может быть пустым):

64, 76, 54, 68

Как насчет получения списка случайных паролей до тех пор, пока первый сгенерированный пароль не будет содержать символ `!`?

Что ж, основываясь на приведенном ранее помощнике, мы можем сделать это вот так:

```
List<String> result = Stream.generate(Main::randomPassword)
    .takeWhile(s -> s.contains("!"))
    .collect(Collectors.toList());
```

Один из вероятных результатов может быть получен следующим образом (он также может быть пустым):

```
0!dac!3c, 2!$!b2ac, 1d12ba1!
```

Теперь допустим, что мы имеем неупорядоченный поток целых чисел. Следующий фрагмент кода содержит подмножество элементов, которые меньше или равны 10:

```
Set<Integer> setOfInts = new HashSet<>(  
    Arrays.asList(1, 4, 3, 52, 9, 40, 5, 2, 31, 8));  
  
List<Integer> result = setOfInts.stream()  
    .takeWhile(i -> i <= 10)  
    .collect(Collectors.toList());
```

Один из возможных результатов выглядит следующим образом (помните, что для неупорядоченного потока результат является недетерминированным):

```
1, 3, 4
```

Отбрасывать до тех пор, пока предикат возвращает *true*

Начиная с JDK 9, у нас также есть метод `Stream.dropWhile(Predicate<? super T> predicate)`. Он противоположен методу `takeWhile()`. Вместо того чтобы принимать элементы до тех пор, пока заданный предикат не вернет `false`, этот метод отбрасывает элементы до тех пор, пока заданный элемент не вернет `false`, и затем включает остальные элементы в возвращаемый поток:

- ◆ Если поток упорядочен, данный метод возвращает поток, состоящий из оставшихся элементов этого потока после удаления *наибольшего префикса элементов*, совпадающих с заданным предикатом.
- ◆ Если поток не упорядочен и некоторые (но не все) элементы этого потока совпадают с заданным предикатом, то поведение этой операции не детерминировано; она свободна отбросить любое подмножество совпадающих элементов (которое включает пустое множество).

В случае упорядоченного потока *наибольший префикс элементов* — это непрерывная последовательность элементов потока, совпадающих с заданным предикатом.

Например, давайте соберем 5 целых чисел после отбрасывания первых 10:

```
List<Integer> result = IntStream  
    .iterate(1, i -> i + 1)  
    .dropWhile(i -> i <= 10)  
    .limit(5)  
    .boxed()  
    .collect(Collectors.toList());
```

Это всегда дает следующий результат:

```
11, 12, 13, 14, 15
```

В качестве альтернативы мы можем получить список из пяти случайных четных целых чисел, превышающих 50 (по крайней мере, это то, что, по нашему мнению, код делает):

```
List<Integer> result = new Random().ints(1, 100)
    .filter(i -> i % 2 == 0)
    .dropWhile(i -> i < 50)
    .limit(5)
    .boxed()
    .collect(Collectors.toList());
```

Один из возможных результатов выглядит следующим образом:

78, 16, 4, 94, 26

Но почему там 16 и 4? Они являются четными, но не больше 50! Все дело в том, что они находятся там, потому что появились после первого элемента, который заставил предикат вернуть `false`. В общих чертах, мы отбрасываем значения до тех пор, пока они меньше 50 (`dropWhile(i -> i < 50)`). Значение 78 заставит этот предикат вернуть `false`, и поэтому метод `dropWhile()` прекратит свою работу. Далее все сгенерированные элементы включаются в результат до тех пор, пока не сработает предел `limit(5)`.

Давайте рассмотрим еще одну похожую ловушку. Получим список из пяти случайных паролей, содержащих символ `!` (по крайней мере, это то, что по нашему мнению, код делает):

```
List<String> result = Stream.generate(Main::randomPassword)
    .dropWhile(s -> !s.contains("!"))
    .limit(5)
    .collect(Collectors.toList());
```

Один из возможных результатов выглядит следующим образом:

`bab2!3dd, c2@$!acc, $c1c@cb@, !b2!$cdc, #b103c21`

Опять же мы видим пароли, которые не содержат символ `!`.

Пароль `bab2!3dd` заставит наш предикат вернуть `false` и в итоге вернет финальный результат (список). Четыре сгенерированных пароля добавляются к результату, на которые метод `dropWhile()` уже не влияет.

Теперь допустим, что мы имеем неупорядоченный поток целых чисел. Следующий фрагмент кода отбрасывает подмножество элементов, которые меньше или равны 10, и оставляет остальные:

```
Set<Integer> setOfInts = new HashSet<>()
    Arrays.asList(5, 42, 3, 2, 11, 1, 6, 55, 9, 7));

List<Integer> result = setOfInts.stream()
    .dropWhile(i -> i <= 10)
    .collect(Collectors.toList());
```

Один из возможных результатов выглядит следующим образом (помните, что для неупорядоченного потока результат является недетерминированным):

55, 7, 9, 42, 11

С одной стороны, если все элементы совпадают с заданным предикатом, то метод `takeWhile()` берет, а метод `dropWhile()` отбрасывает все элементы (не имеет значения, упорядочен поток или не упорядочен). С другой стороны, если ни один из элементов не совпадает с заданным предикатом, то метод `takeWhile()` ничего не берет (возвращает пустой поток), а метод `dropWhile()` ничего не отбрасывает (возвращает поток).



Избегайте использования методов `takeWhile()/dropWhile()` в контексте параллельных потоков, поскольку эти операции являются дорогостоящими, в особенности для упорядоченных потоков. Если это подходит для данного случая, то просто уберите упорядочивающее ограничение посредством метода `BaseStream.unordered()`.

182. Отображение элементов потока в новый поток

Отображение элементов потока — это промежуточная операция, которая используется для *трансформирования* этих элементов в новую их версию путем применения заданной функции к каждому элементу и накопления результатов в новом потоке (например, трансформирование потока `Stream<String>` в поток `Stream<Integer>` или потока `Stream<String>` в еще один поток `Stream<String>` и т. д.).

Использование метода *Stream.map()*

Мы вызываем метод `Stream.map(Function<? super T,? extends R> mapper)`, чтобы применить функцию `mapper` для каждого элемента потока. В результате образуется новый поток. Он не модифицирует исходный поток.

Допустим, что мы имеем следующий ниже класс `Melon`:

```
public class Melon {
    private String type;
    private int weight;

    // конструкторы, геттеры, сеттеры, equals(),
    // hashCode(), toString() опущены для краткости
}
```

Мы также должны допустить, что у нас есть список объектов класса `Melon`, `List<Melon>`:

```
List<Melon> melons = Arrays.asList(new Melon("Gac", 2000),
    new Melon("Hemi", 1600), new Melon("Gac", 3000),
    new Melon("Apollo", 2000), new Melon("Horned", 1700));
```

Далее мы хотим извлечь только названия дынь в еще один список, `List<String>`.

В выполнении этой операции мы можем опереться на метод `map()`:

```
List<String> melonNames = melons.stream()
    .map(Melon::getType)
    .collect(Collectors.toList());
```

Результат будет содержать следующие ниже сорта дынь:

Gac, Hemi, Gac, Apollo, Horned

Схема на рис. 9.7 показывает принцип работы метода `map()` для этого примера.

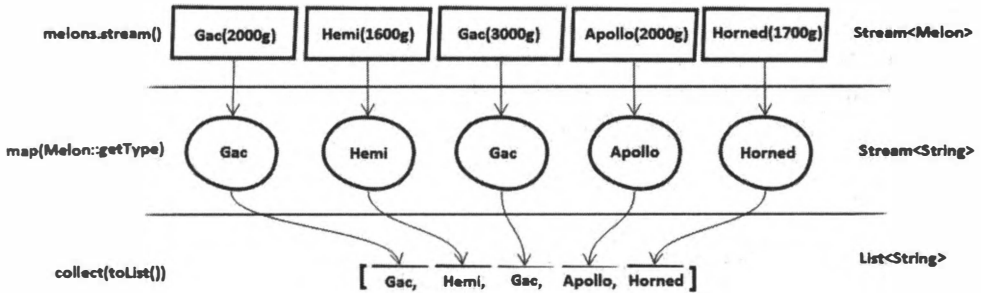


Рис. 9.7

Таким образом, метод `map()` получает поток `Stream<Melon>` и выводит поток `Stream<String>`. Каждая дыня проходит через метод `map()`, и этот метод извлекает сорт дыни (т. е. значение типа `String`) и сохраняет его в другом потоке.

Точно так же мы можем извлечь вес дыни. Поскольку веса являются целыми числами, метод `map()` вернет `Stream<Integer>`:

```
List<Integer> melonWeights = melons.stream()
    .map(Melon::getWeight)
    .collect(Collectors.toList());
```

Результат будет содержать следующие ниже веса:

2000, 1600, 3000, 2000, 1700



Помимо метода `map()`, класс `Stream` также предусматривает разновидности для примитивов, такие как `mapToInt()`, `mapToLong()` и `mapToDouble()`. Эти методы возвращают специализации потока для примитивных типов `int` (`IntStream`), `long` (`LongStream`) и `double` (`StreamDouble`).

Хотя метод `map()` может отображать элементы потока в новый поток посредством `Function`, не следует самонадеянно думать, что мы можем сделать так:

```
List<Melon> lighterMelons = melons.stream()
    .map(m -> m.setWeight(m.getWeight() - 500))
    .collect(Collectors.toList());
```

Этот код не будет работать/компилироваться, потому что метод `setWeight()` возвращает `void`. Для того чтобы заставить его работать, нам нужно вернуть объект

Melon, но это означает, что мы должны добавить некоторый формальный код (например, `return`):

```
List<Melon> lighterMelons = melons.stream()
    .map(m -> {
        m.setWeight(m.getWeight() - 500);

        return m;
    })
    .collect(Collectors.toList());
```

Что вы думаете об искушении применить метод `peek()`? Дело в том, что метод `peek()` означает *"посмотреть, но не трогать"*, однако его можно использовать для мутирования состояния, как показано ниже:

```
List<Melon> lighterMelons = melons.stream()
    .peek(m -> m.setWeight(m.getWeight() - 500))
    .collect(Collectors.toList());
```

Результат будет содержать следующие дыни (выглядит неплохо):

```
Gac(1500g), Hemi(1100g), Gac(2500g), Apollo(1500g), Horned(1200g)
```

Так понятнее, нежели использование метода `map()`. Вызов метода `setWeight()` является четким сигналом о том, что мы планируем мутировать состояние, но в документации говорится о том, что потребитель `Consumer`, передаваемый в метод `peek()`, должен представлять собой *невмешивающееся действие* (не должен модифицировать источник потоковых данных).

Для последовательных потоков (таких как приведенный выше) нарушение этого ожидания может держаться под контролем без побочных эффектов; однако в случае конвейеров параллельных потоков проблема может усложниться.

Действие может быть вызвано в любое время и в любой нити исполнения, в которой элемент становится доступным операции, вышестоящей в потоке, поэтому если действие модифицирует совместное состояние, то оно отвечает за обеспечение необходимой синхронизации.

В качестве общего правила, хорошенько подумайте перед тем, как использовать `peek()` для мутирования состояния. Кроме того, имейте в виду, что это практическое решение является дискуссионным и подпадает под категорию плохой практики или даже антишаблонов.

Использование метода `Stream.flatMap()`

Итак, мы только что видели, метод `map()` знает, как обертывать последовательность элементов в поток `Stream`.

Это означает, что метод `map()` может создавать такие потоки, как `Stream<String[]>`, `Stream<List<String>>`, `Stream<Set<String>>` или даже `Stream<Stream<R>>`.

Но проблема заключается в том, что этими типами потоков невозможно манипулировать успешно (либо, как мы ожидали) с помощью потоковых операций, таких как `sum()`, `distinct()`, `filter()` и т. д.

Для примера рассмотрим следующий массив объектов класса `Melon`:

```
Melon[][] melonsArray = {
    {new Melon("Gac", 2000), new Melon("Hemi", 1600)},
    {new Melon("Gac", 2000), new Melon("Apollo", 2000)},
    {new Melon("Horned", 1700), new Melon("Hemi", 1600)}
};
```

Мы можем взять этот массив и обернуть его в поток посредством метода `Arrays.stream()`, как показано в следующем фрагменте кода:

```
Stream<Melon[]> streamOfMelonsArray = Arrays.stream(melonsArray);
```



Существует много других способов получения потока массивов. Например, если у нас есть строковое значение `s`, то `map(s -> s.split(""))` вернет `Stream<String[]>`.

Далее мы можем подумать, что для получения разных экземпляров класса `Melon` достаточно вызвать метод `distinct()` следующим образом:

```
streamOfMelonsArray
    .distinct()
    .collect(Collectors.toList());
```

Но это не сработает, потому что метод `distinct()` не будет искать отдельный объект `Melon`; вместо этого он займется поиском отдельного массива объектов `Melon[]`, потому что именно они имеются в потоке.

Более того, результат, который был возвращен в этом случае, имеет тип `Stream<Melon[]>`, а не тип `Stream<Melon>`. Финальный результат соберет `Stream<Melon[]>` в `List<Melon[]>`.

Как устранить эту проблему?

Мы можем попытаться применить метод `Arrays.stream()` для того, чтобы конвертировать массив объектов `Melon[]` в поток `Stream<Melon>`:

```
streamOfMelonsArray
    .map(Arrays::stream) // Stream<Stream<Melon>>
    .distinct()
    .collect(Collectors.toList());
```

Опять же, метод `map()` не станет делать то, что мы предполагаем.

Сначала вызов метода `Arrays.stream()` вернет `Stream<Melon>` из каждого заданного `Melon[]`. Однако `map()` возвращает поток элементов, и поэтому он будет обортывать результаты применения метода `Arrays.stream()` в поток `Stream`. В итоге он даст `Stream<Stream<Melon>>`.

Таким образом, на этот раз метод `distinct()` пытается обнаружить отдельные элементы потока `Stream<Melon>` (рис. 9.8).

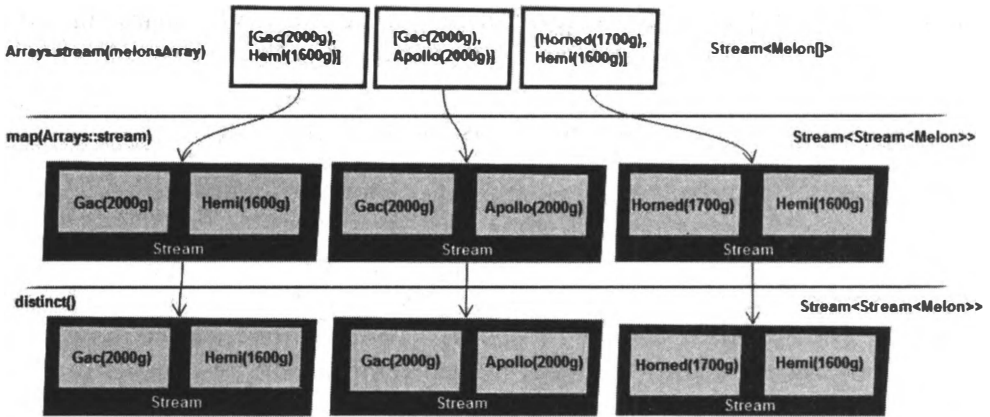


Рис. 9.8

Для того чтобы устранить эту проблему, мы должны опереться на метод `flatMap()`. Схема на рис. 9.9 демонстрирует принцип работы метода `flatMap()` внутри.

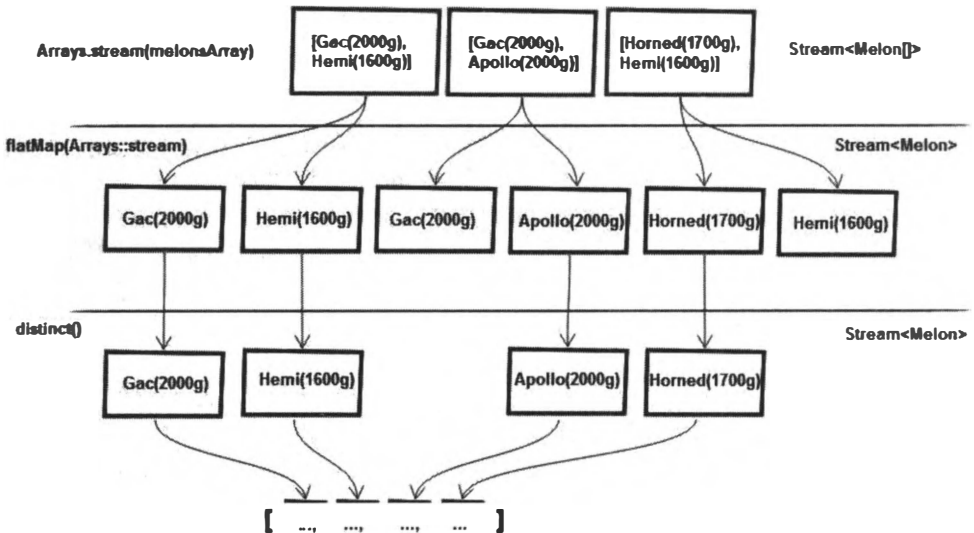


Рис. 9.9

В отличие от метода `map()`, этот метод возвращает поток, сглаживая все разделенные потоки. Таким образом, все массивы окажутся в одном потоке:

```
streamOfMelonsArray
    .flatMap(Arrays::stream) // Stream<Melon>
    .distinct()
    .collect(Collectors.toList());
```

Результат будет содержать разные дыни в зависимости от имплементации метода `Melon.equals()`:

```
Gac(2000g), Hemi(1600g), Apollo(2000g), Horned(1700g)
```

Теперь давайте рассмотрим еще одну задачу и начнем со списка `List<List<String>>` следующим образом:

```
List<List<String>> melonLists = Arrays.asList(
    Arrays.asList("Gac", "Cantaloupe"),
    Arrays.asList("Hemi", "Gac", "Apollo"),
    Arrays.asList("Gac", "Hemi", "Cantaloupe"),
    Arrays.asList("Apollo"),
    Arrays.asList("Horned", "Hemi"),
    Arrays.asList("Hemi"));
```

Мы пытаемся получить разные названия дынь из этого списка. Если обернуть массив в поток можно посредством метода `Arrays.stream()`, то для коллекции у нас есть метод `Collection.stream()`. Поэтому первая попытка может выглядеть следующим образом:

```
melonLists.stream()
    .map(Collection::stream)
    .distinct();
```

Но исходя из предыдущей задачи, мы уже знаем, что она не сработает, потому что `map()` вернет `Stream<Stream<String>>`.

Решение обеспечивается методом `flatMap()`:

```
List<String> distinctNames = melonLists.stream()
    .flatMap(Collection::stream)
    .distinct()
    .collect(Collectors.toList());
```

Результат выглядит так:

```
Gac, Cantaloupe, Hemi, Apollo, Horned
```



Помимо метода `flatMap()`, класс `Stream` также предусматривает разновидности для примитивов, такие как `flatMapToInt()`, `flatMapToLong()` и `flatMapToDouble()`. Эти методы возвращают специализации потока для примитивных типов `int` (`IntStream`), `long` (`LongStream`) и `double` (`StreamDouble`).

183. Отыскание элементов в потоке

Помимо использования метода `filter()`, который позволяет фильтровать элементы потока по предикату, мы можем отыскать элемент в потоке посредством методов `anyFirst()` и `findFirst()`.

Допустим, что у нас есть следующий ниже список, завернутый в поток:

```
List<String> melons = Arrays.asList(
    "Gac", "Cantaloupe", "Hemi", "Gac", "Gac",
    "Hemi", "Cantaloupe", "Horned", "Hemi", "Hemi");
```

Метод *findAny*

Метод `findAny()` возвращает произвольный (недетерминированный) элемент из потока. Например, следующий фрагмент кода вернет элемент из предыдущего списка:

```
Optional<String> anyMelon = melons.stream()
    .findAny();

if (!anyMelon.isEmpty()) {
    System.out.println("Любая дыня: " + anyMelon.get());
} else {
    System.out.println("Никаких дынь не найдено");
}
```



Обратите внимание, что нет никакой гарантии, что при каждом исполнении код будет возвращать одинаковый элемент. Это утверждение особенно верно в случае параллелизованного потока.

Мы также можем комбинировать метод `findAny()` с другими операциями. Вот пример:

```
String anyApollo = melons.stream()
    .filter(m -> m.equals("Apollo"))
    .findAny()
    .orElse("отсутствует");
```

На этот раз результат будет отрицательным. В списке нет дыни `Apollo`, и поэтому операция `filter()` создаст пустой поток. Кроме того, метод `findAny()` также вернет пустой поток, поэтому метод `orElse()` вернет финальный результат в виде указанной в нем строки "отсутствует".

Метод *findFirst*

Если метод `findAny()` возвращает из потока любой элемент, то метод `findFirst()` возвращает первый элемент из потока. Очевидно, что этот метод полезен, когда нас интересует только первый элемент потока (например, победитель конкурса должен быть первым элементом в отсортированном списке соперников).



Тем не менее, если поток не имеет *порядка появления*, то может быть возвращен любой элемент. Согласно документации, *потоки могут не иметь определенный порядок появления. Это зависит от источника и промежуточных операций. То же самое правило применимо и в параллелизме.*

А пока допустим, что нам нужна первая дыня в списке:

```
Optional<String> firstMelon = melons.stream()
    .findFirst();

if (!firstMelon.isEmpty()) {
    System.out.println("Первая дыня: " + firstMelon.get());
}
```

```
} else {
    System.out.println("Никаких дынь не найдено");
}
```

Результат будет следующим:

```
First melon: Gac
```

Мы можем комбинировать метод `findFirst()` и с другими операциями. Вот пример:

```
String firstApollo = melons.stream()
    .filter(m -> m.equals("Apollo"))
    .findFirst()
    .orElse("отсутствует");
```

На этот раз результатом будет "отсутствует", т. к. `filter()` создаст пустой поток.

Ниже приведена еще одна задача с целыми числами (обратите внимание на комментарии справа, чтобы разобраться в последствиях операций):

```
List<Integer> ints = Arrays.asList(4, 8, 4, 5, 5, 7);

int result = ints.stream()
    .map(x -> x * x - 1)           // 23, 63, 23, 24, 24, 48
    .filter(x -> x % 2 == 0)      // 24, 24, 48
    .findFirst()                 // 24
    .orElse(-1);
```

184. Сопоставление элементов в потоке

Для сопоставления тех или иных элементов в потоке мы можем опираться на следующие методы:

- ◆ `anyMatch()`;
- ◆ `noneMatch()`;
- ◆ `allMatch()`.

Все эти методы принимают `Predicate` в качестве аргумента и при сопоставлении с ним получают результат типа `boolean`.



Эти три операции основаны на укорочении вычислений. Другими словами, эти методы могут вернуться до того, как мы обработаем весь поток целиком. Например, если метод `allMatch()` дает `false` (оценивает заданный предикат как `false`), то нет причин продолжать. Финальный результат будет равен `false`.

Допустим, что у нас есть следующий список, завернутый в поток:

```
List<String> melons = Arrays.asList(
    "Gac", "Cantaloupe", "Hemi", "Gac", "Gac", "Hemi",
    "Cantaloupe", "Horned", "Hemi", "Hemi");
```

Теперь попробуем ответить на вопросы:

- ◆ Совпадает ли элемент со строковым значением `Gac`? Давайте посмотрим это в следующем ниже фрагменте кода:

```
boolean isAnyGac = melons.stream()
    .anyMatch(m -> m.equals("Gac")); // true
```

- ◆ Совпадает ли элемент со строковым значением `Apollo`? Давайте посмотрим это в следующем ниже фрагменте кода:

```
boolean isAnyApollo = melons.stream()
    .anyMatch(m -> m.equals("Apollo")); // false
```

Или в качестве обобщающего вопроса: имеется ли в потоке элемент, который совпадает с заданным предикатом?

- ◆ Не совпадают ли все элементы со строковым значением `Gac`? Давайте посмотрим это в следующем ниже фрагменте кода:

```
boolean isNoneGac = melons.stream()
    .noneMatch(m -> m.equals("Gac")); // false
```

- ◆ Не совпадают ли все элементы со строковым значением `Apollo`? Давайте посмотрим это в следующем ниже фрагменте кода:

```
boolean isNoneApollo = melons.stream()
    .noneMatch(m -> m.equals("Apollo")); // true
```

Или в качестве обобщающего вопроса: отсутствуют ли в потоке элементы, которые совпадают с заданным предикатом?

- ◆ Совпадают ли все элементы со строковым значением `Gac`? Давайте посмотрим это в следующем ниже фрагменте кода:

```
boolean areAllGac = melons.stream()
    .allMatch(m -> m.equals("Gac")); // false
```

- ◆ Все ли элементы больше 2? Давайте посмотрим это в следующем ниже фрагменте кода:

```
boolean areAllLargerThan2 = melons.stream()
    .allMatch(m -> m.length() > 2);
```

Или в качестве обобщающего вопроса: все ли элементы в потоке совпадают с заданным предикатом?

185. Сумма, максимум и минимум в потоке

Допустим, что мы имеем класс `Melon`:

```
public class Melon {
    private String type;
    private int weight;

    // конструкторы, геттеры, сеттеры, equals(),
    // hashCode(), toString() опущены для краткости
}
```

Давайте также предположим, что у нас есть следующий ниже список объектов класса `Melon`, обернутый в поток:

```
List<Melon> melons = Arrays.asList(new Melon("Gac", 2000),
    new Melon("Hemi", 1600), new Melon("Gac", 3000),
    new Melon("Apollo", 2000), new Melon("Horned", 1700));
```

Поработаем с классом `Melon`, используя терминальные операции `sum()`, `min()` и `max()`.

Терминальные операции `sum()`, `min()` и `max()`

Теперь давайте объединим элементы этого потока для того, чтобы выразить следующие запросы:

- ◆ Как рассчитать общий вес дынь (`sum()`)?
- ◆ Какая дыня является самой тяжелой (`max()`)?
- ◆ Какая дыня является самой легкой (`min()`)?

Для того чтобы вычислить общий вес дынь, нам нужно просуммировать все веса. Для примитивных специализаций потока (`IntStream`, `LongStream` и т. д.) API потоков Java предусматривает терминальную операцию `sum()`. Как следует из названия, этот метод суммирует элементы потока:

```
int total = melons.stream()
    .mapToInt (Melon::getWeight)
    .sum();
```

Помимо метода `sum()` у нас также есть терминальные операции `max()` и `min()`. Очевидно, что метод `max()` возвращает максимальное значение потока, а метод `min()` — его противоположность:

```
int max = melons.stream()
    .mapToInt (Melon::getWeight)
    .max()
    .orElse(-1);
```

```
int min = melons.stream()
    .mapToInt (Melon::getWeight)
    .min()
    .orElse(-1);
```



Операции `max()` и `min()` возвращают значение типа `OptionalInt` (например, `OptionalLong`). Если максимум или минимум нельзя вычислить (например, в случае пустого потока), то мы возвращаем `-1`. Поскольку мы работаем с весами и с положительными числами по их природе, возвращение `-1` имеет смысл. Но не берите это за правило. В зависимости от случая, может потребоваться вернуть другое значение, или же, возможно, будет лучше использовать методы `orElseGet()` либо `orElseThrow()`.

По поводу непримитивных специализаций обратитесь к разд. 188 "Коллекторы суммирования" далее в этой главе.

В следующем разделе займемся редукцией.

Редукция

Методы `sum()`, `max()` и `min()` именуются частными случаями редукции. Под *редукцией* мы понимаем абстракцию, основанную на двух главных положениях:

- ♦ взять начальное значение (`T`);
- ♦ взять бинарный оператор `BinaryOperator<T>`, для того чтобы объединить два элемента и получить новое значение.

Редукции выполняются посредством терминальной операции `reduce()`, которая соблюдает эту абстракцию и определяет две сигнатуры (вторая не использует начальное значение):

- ♦ `T reduce(T identity, BinaryOperator<T> accumulator);`
- ♦ `Optional<T> reduce(BinaryOperator<T> accumulator).`

Учитывая сказанное, мы можем опереться на терминальную операцию `reduce()` для вычисления суммы элементов следующим образом (начальное значение равно 0, и лямбда-выражение равно `(m1, m2) -> m1 + m2`):

```
int total = melons.stream()
    .map(Melon::getWeight)
    .reduce(0, (m1, m2) -> m1 + m2);
```

Схема на рис. 9.10 демонстрирует принцип работы операции `reduce()`.

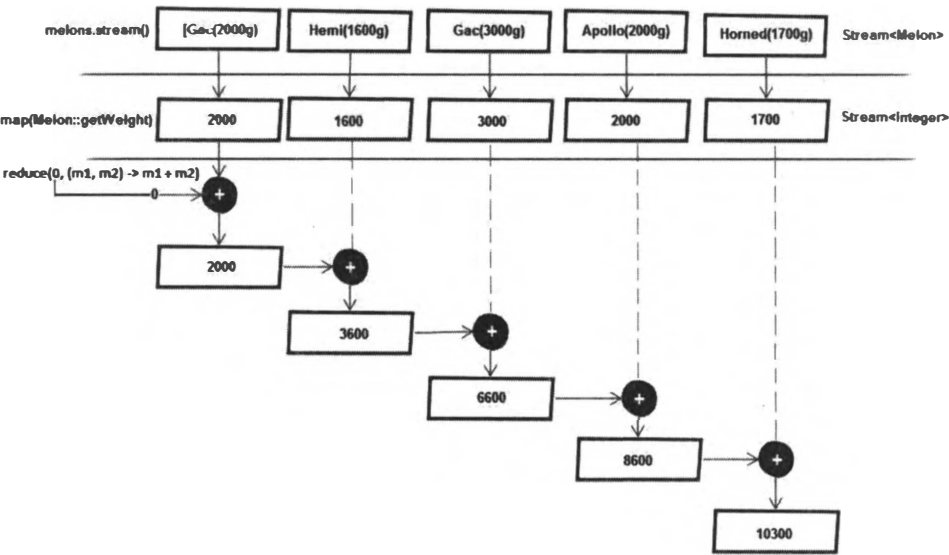


Рис. 9.10

Так как же работает операция `reduce()` ?

Давайте взглянем на следующие ниже шаги, чтобы в этом разобраться.

1. Сначала, в качестве первого параметра лямбда-выражения (`m1`) используется 0, а 2000 потребляется из потока и используется в качестве второго параметра (`m2`). `0 + 2000` дает 2000, которое становится новым накопленным значением.

2. Затем лямбда-выражение вызывается снова уже с накопленным значением и следующим элементом потока, равным 1600, вычислив новое накопленное значение 3600.
3. Двигаясь вперед, лямбда-выражение снова вызывается с накопленным значением и следующим элементом 3000, произведя 6600.
4. Если мы еще раз шагнем вперед, то лямбда-выражение опять будет вызвано с накопленным значением и следующим элементом 2000, произведя 8600.
5. Наконец, лямбда-выражение вызывается с 8600 и последним элементом потока 1700, произведя финальное значение 10 300.

Также можно рассчитать максимум и минимум:

```
int max = melons.stream()
    .map(Melon::getWeight)
    .reduce(Integer::max)
    .orElse(-1);
```

```
int min = melons.stream()
    .map(Melon::getWeight)
    .reduce(Integer::min)
    .orElse(-1);
```

Преимущество использования метода `reduce()` состоит в том, что мы можем легко изменить вычисление, просто передав другое лямбда-выражение. Например, мы можем быстро заменить сумму произведением, как показано в следующем примере:

```
List<Double> numbers = Arrays.asList(1.0d, 5.0d, 8.0d, 10.0d);
```

```
double total = numbers.stream()
    .reduce(1.0d, (x1, x2) -> x1 * x2);
```

Тем не менее обратите внимание на случаи, которые могут привести к нежелательным результатам. Например, если мы хотим вычислить среднее гармоническое из заданных чисел, то стандартного частного случая *редукции* не существует, и поэтому мы можем опереться только на `reduce()` так:

```
List<Double> numbers = Arrays.asList(1.0d, 5.0d, 8.0d, 10.0d);
```

Формула гармонического среднего выглядит следующим образом:

$$H = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \dots + \frac{1}{x_n}} = \frac{n}{\sum_{i=1}^n \frac{1}{x_i}} = \left(\frac{\sum_{i=1}^n (x_i)^{-1}}{n} \right)^{-1}.$$

В нашем случае n — это размер списка, а $H = 2,80701$. Наивное использование метода `reduce()` будет выглядеть следующим образом:

```
double hm = numbers.size() / numbers.stream()
    .reduce((x1, x2) -> (1.0d / x1 + 1.0d / x2))
    .orElseThrow();
```

В результате будет получено число 3,49809.

Это объясняется тем, как мы выразили вычисление. На первом шаге мы вычисляем $1,0/1,0 + 1,0/5,0 = 1,2$. Затем мы ожидаем, что будет вычисляться $1,2 + 1,0/1,8$, но на самом деле вычисляется $1,0/1,2 + 1,0/1,8$. Совершенно очевидно, что это не тот результат, который нам нужен.

Мы можем исправить это вычисление с помощью метода `mapToDouble()`:

```
double hm = numbers.size() / numbers.stream()
    .mapToDouble(x -> 1.0d / x)
    .reduce((x1, x2) -> (x1 + x2))
    .orElseThrow();
```

В результате будет получен ожидаемый результат, т. е. 2.80701.

186. Сбор результатов потока

Допустим, что мы имеем следующий класс `Melon`:

```
public class Melon {
    private String type;
    private int weight;

    // конструкторы, геттеры, сеттеры, equals(),
    // hashCode(), toString() опущены для краткости
}
```

Давайте также предположим, что у нас есть список объектов класса `Melon`:

```
List<Melon> melons = Arrays.asList(new Melon("Crenshaw", 2000),
    new Melon("Hemi", 1600), new Melon("Gac", 3000),
    new Melon("Apollo", 2000), new Melon("Horned", 1700),
    new Melon("Gac", 3000), new Melon("Cantaloupe", 2600));
```

В типичной ситуации потоковый конвейер заканчивается сводкой элементов в потоке. Другими словами, нам нужно собрать результаты в структуру данных, такую как список `List`, множество `Set` или отображение `Map` (и их спутники).

В выполнении этой операции мы можем опереться на метод `Stream.collect(Collector<? super T,A,R> collector)`, который получает один аргумент, представляющий `java.util.stream.Collector` или определяемый пользователем коллектор.

К наиболее известным коллекторам относятся следующие:

- ◆ `toList()`;
- ◆ `toSet()`;
- ◆ `toMap()`;
- ◆ `toCollection()`.

Их имена говорят сами за себя. Давайте рассмотрим несколько примеров.

- ◆ Отфильтровать дыни, которые тяжелее 1000 г, и собрать результат в список `List` посредством методов `toList()` и `toCollection()`:

```
List<Integer> resultToList = melons.stream()
    .map(Melon::getWeight)
    .filter(x -> x >= 1000)
    .collect(Collectors.toList());

List<Integer> resultToList = melons.stream()
    .map(Melon::getWeight)
    .filter(x -> x >= 1000)
    .collect(Collectors.toCollection(ArrayList::new));
```

Аргументом метода `toCollection()` является поставщик `Supplier`, обеспечивающий новую пустую коллекцию, в которую будут вставляться результаты.

- ◆ Отфильтровать дыни, которые тяжелее 1000 г, и собрать результат без повторов в множество `Set` посредством методов `toSet()` и `toCollection()`:

```
Set<Integer> resultToSet = melons.stream()
    .map(Melon::getWeight)
    .filter(x -> x >= 1000)
    .collect(Collectors.toSet());

Set<Integer> resultToSet = melons.stream()
    .map(Melon::getWeight)
    .filter(x -> x >= 1000)
    .collect(Collectors.toCollection(HashSet::new));
```

- ◆ Отфильтровать дыни, которые тяжелее 1000 г, собрать результат без повторов и отсортировать в возрастающем порядке в множество `Set` посредством метода `toCollection()`:

```
Set<Integer> resultToSet = melons.stream()
    .map(Melon::getWeight)
    .filter(x -> x >= 1000)
    .collect(Collectors.toCollection(TreeSet::new));
```

- ◆ Отфильтровать отдельную дыню и собрать результат в отображение `Map<String, Integer>` посредством коллектора `toMap()`:

```
Map<String, Integer> resultToMap = melons.stream()
    .distinct()
    .collect(Collectors.toMap(Melon::getType, Melon::getWeight));
```

Два аргумента коллектора `toMap()` представляют функцию отображения, которая используется для получения ключей и соответствующих им значений (подвержено исключению `java.lang.IllegalStateException`, выбрасываемому при дублировании ключа, если две дыни имеют одинаковый ключ).

- ◆ Отфильтровать отдельную дыню и собрать результат в отображение `Map<Integer, Integer>` посредством коллектора `toMap()`, используя случайные ключи (подвержено исключению `java.lang.IllegalStateException`, выбрасываемому при дублировании ключа, если генерируются два одинаковых ключа):

```
Map<Integer, Integer> resultToMap = melons.stream()
    .distinct()
    .map(x -> Map.entry(
        new Random().nextInt(Integer.MAX_VALUE), x.getWeight()))
    .collect(Collectors.toMap(Entry::getKey, Entry::getValue));
```

- ◆ Собрать дыни в отображение посредством коллектора `toMap()` и избежать потенциального исключения `java.lang.IllegalStateException`, выбрасываемого при дублировании ключа, выбрав существующее (старое) значение в случае коллизии ключей:

```
Map<String, Integer> resultToMap = melons.stream()
    .collect(Collectors.toMap(Melon::getType, Melon::getWeight,
        (oldValue, newValue) -> oldValue));
```

Последний аргумент коллектора `toMap()` является функцией слияния и используется для разрешения коллизий между значениями, ассоциированными с одинаковым ключом, как передано поставщиком в метод `Map.merge(Object, Object, BiFunction)`.

Очевидно, что выбрать новое значение можно с помощью `(oldValue, newValue) -> newValue`.

- ◆ Поместить предыдущий пример в отсортированное отображение `Map` (например, по весу):

```
Map<String, Integer> resultToMap = melons.stream()
    .sorted(Comparator.comparingInt(Melon::getWeight))
    .collect(Collectors.toMap(Melon::getType, Melon::getWeight,
        (oldValue, newValue) -> oldValue, LinkedHashMap::new));
```

Последний аргумент этой разновидности коллектора `toMap()` представляет поставщика `Supplier`, обеспечивающего новое пустое отображение `Map`, в которое будут вставляться результаты. В данном примере этот поставщик необходим для сохранения порядка после сортировки. Поскольку `HashMap` не гарантирует порядок вставки, мы должны опереться на `LinkedHashMap`.

- ◆ Собрать частоту слов посредством коллектора `toMap()`:

```
String str = "Lorem Ipsum is simply
    Ipsum Lorem not simply Ipsum";

Map<String, Integer> mapOfWords = Stream.of(str)
    .map(w -> w.split("\\s+"))
    .flatMap(Arrays::stream)
    .collect(Collectors.toMap(w -> w.toLowerCase(), w -> 1, Integer::sum));
```



Помимо коллекторов `toList()`, `toMap()` и `toSet()`, класс `Collectors` также представляет наружу коллекторы для немодифицируемых и конкурентных коллекций, такие как методы `toUnmodifiableList()`, `toConcurrentMap()` и т. д.

187. Соединение результатов потока

Допустим, что мы имеем следующий класс `Melon`:

```
public class Melon {
    private String type;
    private int weight;

    // конструкторы, геттеры, сеттеры, equals(),
    // hashCode(), toString() опущены для краткости
}
```

Давайте также предположим, что у нас есть список объектов класса `Melon`:

```
List<Melon> melons = Arrays.asList(new Melon("Crenshaw", 2000),
    new Melon("Hemi", 1600), new Melon("Gac", 3000),
    new Melon("Apollo", 2000), new Melon("Horned", 1700),
    new Melon("Gac", 3000), new Melon("Cantaloupe", 2600));
```

В предыдущей задаче мы говорили об API потоков Java, который встроен в класс `Collectors`. В этой категории у нас также есть связка коллекторов `Collectors.joining()`. Цель этих коллекторов состоит в конкатенации элементов потока в значение типа `String` в *порядке появления*. Опционально эти коллекторы могут использовать разделитель, префикс и суффикс, и поэтому наиболее полной разновидностью метода `joining()` является метод `String joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)`.

Но если же нам нужно лишь конкатенировать названия дынь без разделителя, это как раз тот путь, который нам нужен (только для удобства давайте отсортируем и удалим повторы):

```
String melonNames = melons.stream()
    .map(Melon::getType)
    .distinct()
    .sorted()
    .collect(Collectors.joining());
```

Мы получим вот такой результат:

```
ApolloCantaloupeCrenshawGacHemiHorned
```

Более приятное решение состоит в добавлении разделителя, например запятой и пробела:

```
String melonNames = melons.stream()
    ...
    .collect(Collectors.joining(", "));
```

Получим следующий результат:

```
Apollo, Cantaloupe, Crenshaw, Gac, Hemi, Horned
```

Мы также можем обогатить вывод префиксом и суффиксом:

```
String melonNames = melons.stream()
    ...
    .collect(Collectors.joining(", ",
        "Дыни в наличии: ", " Благодарю!"));
```

В результате мы получим следующее:

```
Дыни в наличии: Apollo, Cantaloupe, Crenshaw, Gac, Hemi, Horned
Благодарю!
```

188. Подытоживающие коллекторы

Допустим, что мы имеем уже хорошо известный нам класс `Melon` (в котором используются `type` и `weight`) и список объектов класса `Melon`:

```
List<Melon> melons = Arrays.asList(new Melon("Crenshaw", 2000),
    new Melon("Hemi", 1600), new Melon("Gac", 3000),
    new Melon("Apollo", 2000), new Melon("Horned", 1700),
    new Melon("Gac", 3000), new Melon("Cantaloupe", 2600));
```

API потоков Java группирует операции подсчета, суммирования, минимума, среднего значения и максимума под термином *"подытоживание"*. Методы, предназначенные для выполнения операций *подытоживания*, находятся в классе `Collectors`.

Мы рассмотрим все эти операции в следующих далее разделах.

Суммирование

Допустим, что мы хотим сложить веса всех дынь. Мы сделали это в *разд. 185 "Сумма, максимум и минимум в потоке"* посредством примитивных специализаций класса `Stream`. А теперь давайте сделаем это посредством коллектора `summingInt`(`ToIntFunction`<? super T> `mapper`):

```
int sumWeightsGrams = melons.stream()
    .collect(Collectors.summingInt(Melon::getWeight));
```

Коллектор `summingInt`(`ToIntFunction`<? super T> `mapper`) представляет собой фабричный метод, который в качестве аргумента берет функцию, способную отобразить объект в подлежащее суммированию значение типа `int`. Возвращается коллектор, который выполняет суммирование посредством метода `collect()`. Схема на рис. 9.11 демонстрирует принцип работы метода `summingInt()`.

При пересечении потока каждый вес (`Melon::getWeight`) отображается в свое число, и это число добавляется в накопитель, начиная с первого значения, т. е. 0.

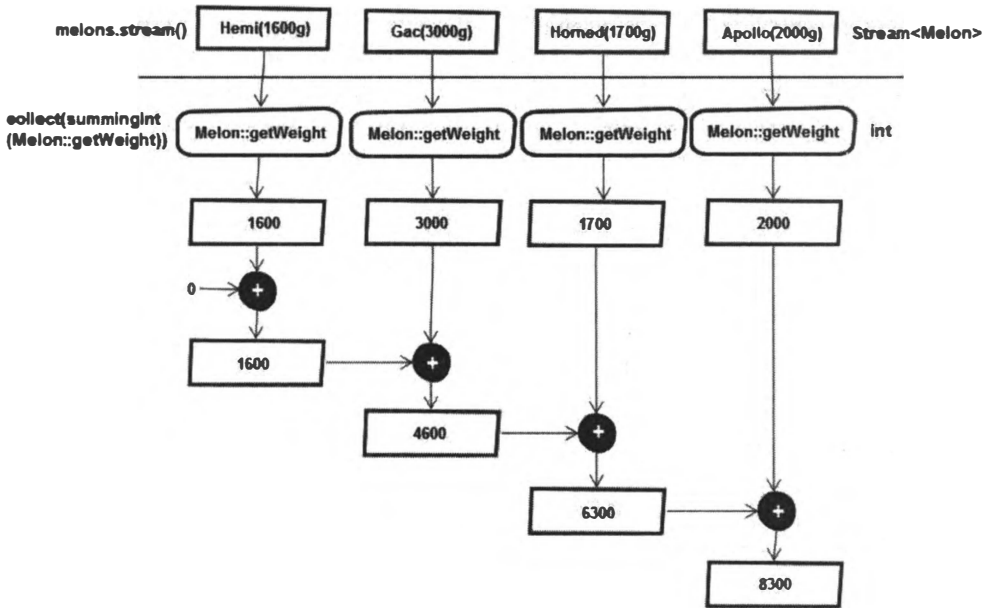


Рис. 9.11

Помимо метода `summingInt()` у нас есть методы `summingLong()` и `summingDouble()`. Как суммировать вес дыни в килограммах? Это можно сделать с помощью метода `summingDouble()` следующим образом:

```
double sumWeightsKg = melons.stream()
    .collect(Collectors.summingDouble(
        m -> (double) m.getWeight() / 1000.0d));
```

Если нам нужен результат просто в килограммах, то мы все равно можем просуммировать в граммах, как показано ниже:

```
double sumWeightsKg = melons.stream()
    .collect(Collectors.summingInt(Melon::getWeight)) / 1000.0d;
```

Поскольку *подытоживания* на самом деле являются *редукциями*, класс `Collectors` также предусматривает коллектор `reducing()`. Очевидно, что этот метод имеет более широкое применение, позволяя нам обеспечить все виды лямбда-выражений посредством трех своих разновидностей:

- ◆ `reducing(BinaryOperator<T> op);`
- ◆ `reducing(T identity, BinaryOperator<T> op);`
- ◆ `reducing(U identity, Function<? super T,? extends U>mapper, BinaryOperator<U> op).`

Аргументы коллектора `reducing()` довольно просты. У нас есть значение тождественности `identity` для редукции (а также значение, которое возвращается, когда входных элементов нет), функция отображения, применяемая к каждому входному значению, и функция, используемая для редукции отображенных значений.

Например, давайте перепишем приведенный выше фрагмент кода с использованием коллектора `reducing()`. Обратите внимание, что мы начинаем сумму с 0, конвертируем ее из граммов в килограммы с помощью функции отображения и редуцируем значения (полученные килограммы) посредством лямбда-выражения:

```
double sumWeightsKg = melons.stream()
    .collect(Collectors.reducing(0.0,
        m -> (double) m.getWeight() / 1000.0d, (m1, m2) -> m1 + m2));
```

В качестве альтернативы мы можем просто преобразовать вес в килограммы в конце:

```
double sumWeightsKg = melons.stream()
    .collect(Collectors.reducing(0,
        m -> m.getWeight(), (m1, m2) -> m1 + m2)) / 1000.0d;
```



Коллектор `reducing()` следует использовать в ситуациях, когда нет подходящего встроенного решения. Думайте о коллекторе `reducing()` как об *обобщенном подытоживании*.

Усреднение

Как насчет вычисления среднего веса дынь?

Для этого у нас есть коллекторы `Collectors.averagingInt()`, `averagingLong()` и `averagingDouble()`:

```
double avgWeights = melons.stream()
    .collect(Collectors.averagingInt(Melon::getWeight));
```

Подсчет

Задача подсчета числа слов во фрагменте текста встречается очень часто и решается посредством метода `count()`:

```
String str = "Lorem Ipsum is simply dummy text ...";
```

```
long numberOfWords = Stream.of(str)
    .map(w -> w.split("\\s+"))
    .flatMap(Arrays::stream)
    .filter(w -> w.trim().length() != 0)
    .count();
```

Но давайте посмотрим, сколько имеется дынь весом в 3000 г в нашем потоке:

```
long nrOfMelon = melons.stream()
    .filter(m -> m.getWeight() == 3000)
    .count();
```

Мы можем применить коллектор, возвращаемый фабричным методом `counting()`:

```
long nrOfMelon = melons.stream()
    .filter(m -> m.getWeight() == 3000)
    .collect(Collectors.counting());
```

Мы также можем применить неуклюжий подход с использованием коллектора `reducing()`:

```
long nrOfMelon = melons.stream()
    .filter(m -> m.getWeight() == 3000)
    .collect(Collectors.reducing(0L, m -> 1L, Long::sum));
```

Максимум и минимум

В разд. 185 *"Сумма, максимум и минимум в потоке"* мы уже вычисляли минимальное и максимальное значения с помощью методов `min()` и `max()`. На этот раз давайте вычислим самую тяжелую и самую легкую дыню посредством коллекторов `Collectors.maxBy()` и `Collectors.minBy()`. Эти методы берут `Comparator` в качестве аргумента для сравнения элементов в потоке и возвращают значение типа `Optional` (значение типа `Optional` будет пустым, если поток пуст):

```
Comparator<Melon> byWeight = Comparator.comparing(Melon::getWeight);
```

```
Melon heaviestMelon = melons.stream()
    .collect(Collectors.maxBy(byWeight))
    .orElseThrow();
```

```
Melon lightestMelon = melons.stream()
    .collect(Collectors.minBy(byWeight))
    .orElseThrow();
```

В этом случае, если поток является пустым, то мы просто выбрасываем исключение `NoSuchElementException`.

Все вместе

Есть ли способ получить общее число, сумму, среднее, минимум и максимум одной унитарной операцией?

Да, есть! Всякий раз, когда нам нужны две или более этих операций, мы можем опереться на коллекторы `Collectors.summarizingInt()`, `summarizingLong()` и `summarizingDouble()`. Указанные методы оборачивают эти операции соответственно в `IntSummaryStatistics`, `LongSummaryStatistics` и `DoubleSummaryStatistics` так:

```
IntSummaryStatistics melonWeightsStatistics = melons
    .stream().collect(Collectors.summarizingInt(Melon::getWeight));
```

Распечатав этот объект, мы получим следующий результат:

```
IntSummaryStatistics{count=7, sum=15900, min=1600,
    average=2271.428571, max=3000}
```

Для каждой из этих операций у нас есть специальные геттеры:

```
int max = melonWeightsStatistics.getMax();
```

Вот и все! Теперь давайте поговорим о группировании элементов потока.

189. Группирование

Допустим, что у нас есть следующий класс `Melon` и список его объектов:

```
public class Melon {
    enum Sugar {
        LOW, MEDIUM, HIGH, UNKNOWN
    }

    private final String type;
    private final int weight;
    private final Sugar sugar;

    // конструкторы, геттеры, сеттеры, equals(),
    // hashCode(), toString() опущены для краткости
}

List<Melon> melons = Arrays.asList(
    new Melon("Crenshaw", 1200),
    new Melon("Gac", 3000), new Melon("Hemi", 2600),
    new Melon("Hemi", 1600), new Melon("Gac", 1200),
    new Melon("Apollo", 2600), new Melon("Horned", 1700),
    new Melon("Gac", 3000), new Melon("Hemi", 2600)
);
```

API потоков Java выставляет наружу ту же функциональность, что и условие SQL `GROUP BY`, делая это посредством коллектора `Collectors.groupingBy()` и его разновидностей.

В то время как условие SQL `GROUP BY` работает с таблицами базы данных, коллектор `Collectors.groupingBy()` оперирует элементами потоков.

Другими словами, коллектор `groupingBy()` способен группировать элементы с некоторыми отличительными характеристиками. До появления потоков и функционального стиля программирования (Java 8) такие операции применялись к коллекциям с помощью связки *спагетти-кода*, который был громоздким, многословным и подверженным ошибкам. Начиная с Java 8, у нас есть *группирующие коллекторы*.

В следующем далее разделе давайте взглянем на одноуровневое и многоуровневое группирование. Начнем с одноуровневого группирования.

Одноуровневое группирование

Все группирующие коллекторы имеют *классификационную функцию* (функцию, которая разносит элементы потока по разным группам). В общих чертах она является экземпляром функционального интерфейса `Function<T, R>`.

Каждый элемент потока (типа `T`) пропускается через эту функцию, и возвращаемым объектом будет *классификатор* (типа `R`). Все возвращаемые типы `R` представляют

ключи (k) отображения `Map<K, V>`, и каждая группа является значением этого отображения `Map<K, V>`.

Другими словами, ключ (k) — это значение, возвращаемое классификационной функцией, а значение (v) — список элементов в потоке, имеющих это классифицированное значение (k). Таким образом, финальный результат имеет тип `Map<K, List<T>>`.

Давайте рассмотрим пример, чтобы пролить свет на это дразнящее мозг объяснение. Пример основан на простейшей разновидности коллектора `groupingBy()`, а именно `groupingBy(Function<? super T,? extends K> classifier)`.

Итак, сгруппируем дыни по сорту:

```
Map<String, List<Melon>> byTypeInList = melons.stream()
    .collect(groupingBy(Melon::getType));
```

Результат будет следующим:

```
{
    Crenshaw = [Crenshaw(1200 g)],
    Apollo = [Apollo(2600 g)],
    Gac = [Gac(3000 g), Gac(1200 g), Gac(3000 g)],
    Hemi = [Hemi(2600 g), Hemi(1600 g), Hemi(2600 g)],
    Horned = [Horned(1700 g)]
}
```

Мы также можем сгруппировать дыни по весу:

```
Map<Integer, List<Melon>> byWeightInList = melons.stream()
    .collect(groupingBy(Melon::getWeight));
```

Результат будет таким:

```
{
    1600 = [Hemi(1600 g)],
    1200 = [Crenshaw(1200 g), Gac(1200 g)],
    1700 = [Horned(1700 g)],
    2600 = [Hemi(2600 g), Apollo(2600 g), Hemi(2600 g)],
    3000 = [Gac(3000 g), Gac(3000 g)]
}
```

Это группирование показано на рис. 9.12. Точнее, это снимок момента, когда `Gac(1200 g)` проходит через классификационную функцию (`Melon::getWeight`).

Итак, в примере классификации дынь ключом является вес дыни, а его значением — список, содержащий все объекты дыни этого веса.



Классификационной функцией может быть ссылка на метод или любое другое лямбда-выражение.

Одной из проблем с предыдущим подходом является наличие нежелательных повторов. Это происходит потому, что значения собираются в список (например,

`3000=[Gac(3000g), Gac(3000g)]`). Но мы можем исправить это, воспользовавшись еще одной разновидностью коллектора `groupBy()`, а именно `groupBy(Function<? super T,? extends K>classifier, Collector<? super T,A,D> downstream)`. На этот раз в качестве второго аргумента мы можем указать требуемый коллектор, нижестоящий в потоке.

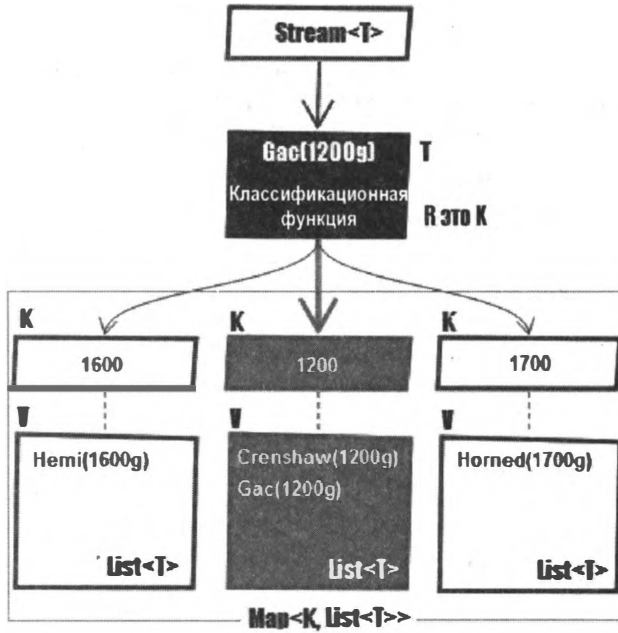


Рис. 9 12

Таким образом, помимо классификационной функции, у нас есть также нижестоящий коллектор.

Если мы хотим отказаться от повторов, то можем использовать метод `Collectors.toSet()` следующим образом:

```
Map<String, Set<Melon>> byTypeInSet = melons.stream()
    .collect(groupingBy(Melon::getType, toSet()));
```

Результат выглядит так:

```
{
  Crenshaw = [Crenshaw(1200 g)],
  Apollo = [Apollo(2600 g)],
  Gac = [Gac(1200 g), Gac(3000 g)],
  Hemi = [Hemi(2600 g), Hemi(1600 g)],
  Horned = [Horned(1700 g)]
}
```

Мы также можем сделать это по весу:

```
Map<Integer, Set<Melon>> byWeightInSet = melons.stream()
    .collect(groupingBy(Melon::getWeight, toSet()));
```

Результат будет следующим:

```
{
  1600 = [Hemi(1600 g)],
  1200 = [Gac(1200 g), Crenshaw(1200 g)],
  1700 = [Horned(1700 g)],
  2600 = [Hemi(2600 g), Apollo(2600 g)],
  3000 = [Gac(3000 g)]
}
```

Конечно, в этом случае можно также применить метод `distinct()`:

```
Map<String, List<Melon>> byTypeInList = melons.stream()
    .distinct()
    .collect(groupingBy(Melon::getType));
```

То же самое будет в случае, если сделать это по весу:

```
Map<Integer, List<Melon>> byWeightInList = melons.stream()
    .distinct()
    .collect(groupingBy(Melon::getWeight));
```

Что ж, теперь повторов больше нет, но результаты не упорядочены. Было бы неплохо упорядочить это отображение по ключам, поэтому отображение по умолчанию `HashMap` оказывается не очень полезным. Если бы вместо отображения по умолчанию `HashMap` мы могли указать древовидное отображение `TreeMap`, задача была бы решена. Мы можем сделать это посредством еще одной разновидности коллектора `groupingBy()`, а именно `groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)`.

Второй аргумент этой разновидности позволяет нам предоставить объект `Supplier`, который в свою очередь предоставляет новое пустое отображение, в которое будут вставляться результаты:

```
Map<Integer, Set<Melon>> byWeightInSetOrdered = melons.stream()
    .collect(groupingBy(Melon::getWeight, TreeMap::new, toSet()));
```

Теперь результат упорядочен:

```
{
  1200 = [Gac(1200 g), Crenshaw(1200 g)],
  1600 = [Hemi(1600 g)],
  1700 = [Horned(1700 g)],
  2600 = [Hemi(2600 g), Apollo(2600 g)],
  3000 = [Gac(3000 g)]
}
```

Мы также можем иметь список `List<Integer>`, содержащий веса 100 дынь:

```
List<Integer> allWeights = new ArrayList<>(100);
```

Мы хотим разделить этот список на 10 списков по 10 весов в каждом. В сущности, мы можем получить это посредством группирования, как показано ниже (мы также можем применить метод `parallelStream()`):

```
final AtomicInteger count = new AtomicInteger();
Collection<List<Integer>> chunkWeights = allWeights.stream()
```

```
.collect(Collectors.groupingBy(c -> count.getAndIncrement() / 10))
.values();
```

А теперь давайте займемся еще одним вопросом. По умолчанию поток `Stream<Melon>` разделен на ряд списков `List<Melon>`. Но что можно сделать для разделения потока `Stream<Melon>` на ряд списков `List<String>`, где каждый список содержит исключительно сорта дынь, а не экземпляры дынь?

Что ж, трансформирование элементов потока обычно является работой метода `map()`. Но внутри коллектора `groupingBy()` этой работой занимается коллектор `Collectors.mapping()` (более подробную информацию можно найти в *разд. 191 "Фильтрующий, сглаживающий и отображающий коллекторы"* далее в этой главе):

```
Map<Integer, Set<String>> byWeightInSetOrdered = melons.stream()
    .collect(groupingBy(Melon::getWeight, TreeMap::new,
        mapping(Melon::getType, toSet())));
```

На этот раз результат является именно таким, каким мы хотели его получить:

```
{
    1200 = [Crenshaw, Gac],
    1600 = [Hemi],
    1700 = [Horned],
    2600 = [Apollo, Hemi],
    3000 = [Gac]
}
```

Так, пока все хорошо! Теперь давайте сосредоточимся на том факте, что две из трех разновидностей коллектора `groupingBy()` принимают коллектор в качестве аргумента (например, `toSet()`). Это может быть любой коллектор. Например, мы можем сгруппировать дыни по сортам и подсчитать их. Для этого будет очень полезен метод `Collectors.counting()` (более подробную информацию можно найти в *разд. 188 "Подытоживающие коллекторы"* ранее в этой главе):

```
Map<String, Long> typesCount = melons.stream()
    .collect(groupingBy(Melon::getType, counting()));
```

Результат будет следующим::

```
{Crenshaw=1, Apollo=1, Gac=3, Hemi=3, Horned=1}
```

Мы также можем сделать это по весу:

```
Map<Integer, Long> weightsCount = melons.stream()
    .collect(groupingBy(Melon::getWeight, counting()));
```

Результат будет таким:

```
{1600=1, 1200=2, 1700=1, 2600=3, 3000=2}
```

Сможем ли мы сгруппировать самые легкие и самые тяжелые дыни по сорту? Конечно, можем! Мы можем сделать это посредством методов `Collectors.minBy()` и `maxBy()`, которые были представлены в *разд. 188 "Подытоживающие коллекторы"*:

```
Map<String, Optional<Melon>> minMelonByType = melons.stream()
    .collect(groupingBy(Melon::getType,
        minBy(comparingInt(Melon::getWeight))));
```

Результат будет выглядеть следующим образом (обратите внимание, что метод `minBy()` возвращает тип `Optional`):

```
{
  Crenshaw = Optional[Crenshaw(1200 g)],
  Apollo = Optional[Apollo(2600 g)],
  Gac = Optional[Gac(1200 g)],
  Hemi = Optional[Hemi(1600 g)],
  Horned = Optional[Horned(1700 g)]
}
```

Мы также можем сделать это посредством метода `maxMelonByType()`:

```
Map<String, Optional<Melon>> maxMelonByType = melons.stream()
  .collect(groupingBy(Melon::getType,
    maxBy(comparingInt(Melon::getWeight))));
```

Результат будет выглядеть так (обратите внимание, что метод `maxBy()` возвращает тип `Optional`):

```
{
  Crenshaw = Optional[Crenshaw(1200 g)],
  Apollo = Optional[Apollo(2600 g)],
  Gac = Optional[Gac(3000 g)],
  Hemi = Optional[Hemi(2600 g)],
  Horned = Optional[Horned(1700 g)]
}
```



Коллекторы `minBy()` и `maxBy()` берут в качестве аргумента компаратор. В этих примерах мы использовали встроенную функцию `Comparator.comparingInt()`. Начиная с JDK 8, класс `java.util.Comparator` был дополнен несколькими новыми компараторами, включая разновидности `thenComparing()` для выстраивания компараторов в цепочку.

Проблема здесь кроется за значениями типа `Optional`, которые должны быть удалены. Вообще, эта категория проблем продолжает адаптировать результат, возвращаемый коллектором, к другому типу.

Что ж, специально для таких операций у нас есть фабричный метод `collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)`. Он берет функцию, которая будет применена к финальному результату коллектора (финишера), нижестоящего в потоке. Его можно использовать следующим образом:

```
Map<String, Integer> minMelonByType = melons.stream()
  .collect(groupingBy(Melon::getType,
    collectingAndThen(minBy(comparingInt(Melon::getWeight)),
      m -> m.orElseThrow().getWeight())));
```

Результат будет следующим:

```
{Crenshaw=1200, Apollo=2600, Gac=1200, Hemi=1600, Horned=1700}
```

Мы также можем использовать метод `maxMelonByType()`:

```
Map<String, Integer> maxMelonByType = melons.stream()
  .collect(groupingBy(Melon::getType,
```



```
collectingAndThen(maxBy(comparingInt(Melon::getWeight)),
    m -> m.orElseThrow().getWeight()));
```

Результат будет таким:

```
{Crenshaw=1200, Apollo=2600, Gac=3000, Hemi=2600, Horned=1700}
```

Мы также можем сгруппировать дыни по сорту в Map<String, Melon[]>. Опять же, в этом мы можем опереться на метод collectingAndThen(), как показано ниже:

```
Map<String, Melon[]> byTypeArray = melons.stream()
    .collect(groupingBy(Melon::getType, collectingAndThen(
        Collectors.toList(), l -> l.toArray(Melon[]::new))));
```

В качестве альтернативы мы можем создать обобщенный коллектор и вызвать его следующим образом:

```
private static <T> Collector<T, ?, T[]>
    toArray(IntFunction<T[]> func) {

    return Collectors.collectingAndThen(
        Collectors.toList(), l -> l.toArray(func.apply(l.size())));
}
```

```
Map<String, Melon[]> byTypeArray = melons.stream()
    .collect(groupingBy(Melon::getType, toArray(Melon[]::new)));
```

Многоуровневое группирование

Ранее мы отметили, что две из трех разновидностей коллектора groupingBy() берут еще один коллектор в качестве аргумента. Более того, мы сказали, что этим коллектором может быть любой. Под любым коллектором мы также подразумеваем и коллектор groupingBy().

Переходя от коллектора groupingBy() к groupingBy(), мы можем достичь *n* уровней группирования или многоуровневого группирования. В общих чертах у нас есть *n* уровней классификационных функций.

Возьмем следующий ниже список объектов класса Melon:

```
List<Melon> melonsSugar = Arrays.asList(
    new Melon("Crenshaw", 1200, HIGH),
    new Melon("Gac", 3000, LOW), new Melon("Hemi", 2600, HIGH),
    new Melon("Hemi", 1600), new Melon("Gac", 1200, LOW),
    new Melon("Cantaloupe", 2600, MEDIUM),
    new Melon("Cantaloupe", 3600, MEDIUM),
    new Melon("Apollo", 2600, MEDIUM), new Melon("Horned", 1200, HIGH),
    new Melon("Gac", 3000, LOW), new Melon("Hemi", 2600, HIGH));
```

Таким образом, каждая дыня имеет свой сорт, вес и показатель уровня сахара. Сначала мы хотим сгруппировать дыни по показателю сахара (низкий, средний, высокий или неизвестный (по умолчанию)). Кроме того, мы хотим сгруппировать дыни

по весу. Это может быть достигнуто посредством следующих двух уровней группирования:

```
Map<Sugar, Map<Integer, Set<String>>> bySugarAndWeight =
melonsSugar.stream()
    .collect(groupingBy (Melon::getSugar,
        groupingBy (Melon::getWeight, TreeMap::new,
            mapping (Melon::getType, toSet()))));
```

Результат выглядит следующим образом:

```
{
  MEDIUM = {
    2600 = [Apollo, Cantaloupe], 3600 = [Cantaloupe]
  },
  HIGH = {
    1200 = [Crenshaw, Horned], 2600 = [Hemi]
  },
  UNKNOWN = {
    1600 = [Hemi]
  },
  LOW = {
    1200 = [Gac], 3000 = [Gac]
  }
}
```

Теперь мы можем сказать, что дыни Crenshaw и Horned весят 1200 г и имеют высокий уровень сахара. У нас также есть дыня Hemi весом 2600 г также с высоким уровнем сахара.

Мы даже можем представить наши данные в виде таблицы (рис. 9.13).

Сахар	НИЗКИЙ	СРЕДНИЙ	ВЫСОКИЙ	НЕИЗВЕСТНЫЙ
Вес				
2600		Apollo Cantaloupe	Hemi	
3600		Cantaloupe		
1200	Gac		Crenshaw Horned	
1600				Hemi
3000	Gac			

Рис. 9.13

Теперь давайте узнаем о разбиении на группы.

190. Разбиение

Разбиение — это вид группирования, который опирается на предикат `Predicate` для разбиения потока на две группы (группу для `true` и группу для `false`). Группа для `true` хранит элементы потока, которые прошли предикат, в то время как группа `false` хранит остальные элементы (элементы, которые предикат не прошел).

Этот предикат представляет *классификационную функцию* и называется *разбивающей функцией*. Поскольку `Predicate` вычисляется как значение типа `boolean`, операция разбиения возвращает отображение `Map<Boolean, V>`.

Допустим, что у нас есть следующий класс `Melon` и список объектов этого класса:

```
public class Melon {
    private final String type;
    private int weight;

    // конструкторы, геттеры, сеттеры, equals(),
    // hashCode(), toString() опущены для краткости
}

List<Melon> melons = Arrays.asList(new Melon("Crenshaw", 1200),
    new Melon("Gac", 3000), new Melon("Hemi", 2600),
    new Melon("Hemi", 1600), new Melon("Gac", 1200),
    new Melon("Apollo", 2600), new Melon("Horned", 1700),
    new Melon("Gac", 3000), new Melon("Hemi", 2600));
```

Разбиение производится посредством коллектора `Collectors.partitioningBy()`. Этот коллектор идет в двух разновидностях, и одна из них получает один аргумент, а именно `partitioningBy(Predicate<? super T> predicate)`.

Например, разбить дыни по весу 2000 г с повторами можно следующим образом:

```
Map<Boolean, List<Melon>> byWeight = melons.stream()
    .collect(partitioningBy(m -> m.getWeight() > 2000));
```

Результат будет таким:

```
{
    false=[Crenshaw(1200g), Hemi(1600g), Gac(1200g), Horned(1700g)],
    true=[Gac(3000g), Hemi(2600g), Apollo(2600g), Gac(3000g), Hemi(2600g)]
}
```



Преимущество разбиения перед фильтрацией состоит в том, что разбиение сохраняет оба списка элементов потока.

Схема на рис. 9.14 демонстрирует принцип работы коллектора `partitioningBy()` внутри.

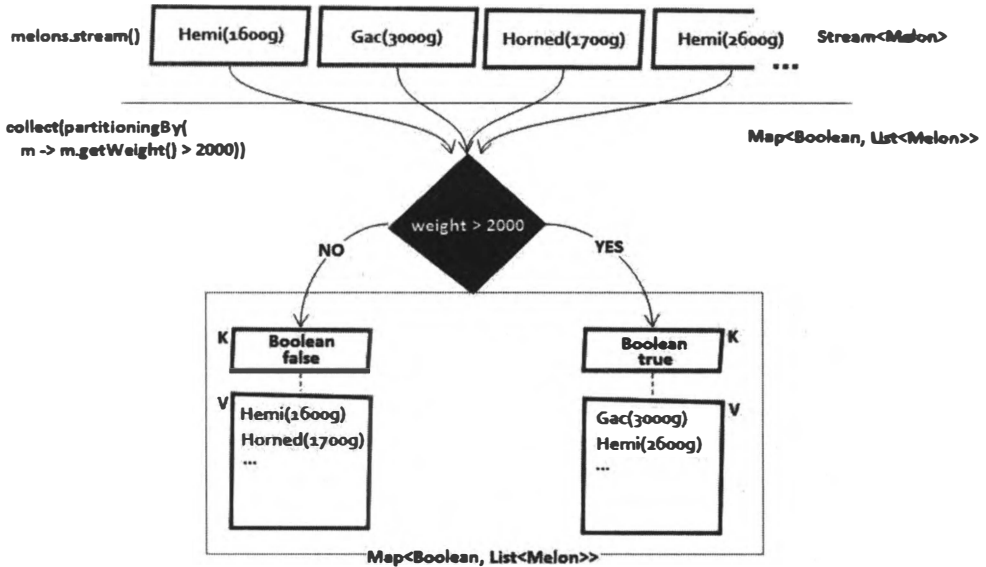


Рис. 9.14

Если мы хотим отклонить повторы, то можем положиться на другие разновидности коллектора `partitioningBy()`, в частности на коллектор `partitioningBy(Predicate<? super T> predicate, Collector<? super T,A,D> downstream)`. Второй аргумент позволяет нам указывать еще один коллектор для имплементирования нижестоящей редукции:

```
Map<Boolean, Set<Melon>> byWeight = melons.stream()
    .collect(partitioningBy(m -> m.getWeight() > 2000, toSet()));
```

Результат не будет содержать повторов:

```
{
    false=[Horned(1700g), Gac(1200g), Crenshaw(1200g), Hemi(1600g)],
    true=[Gac(3000g), Hemi(2600g), Apollo(2600g)]
}
```

Конечно, в данном случае метод `distinct()` тоже справится с этой работой:

```
Map<Boolean, List<Melon>> byWeight = melons.stream()
    .distinct()
    .collect(partitioningBy(m -> m.getWeight() > 2000));
```

Можно использовать и другие коллекторы. Например, мы можем подсчитать элементы из каждой из этих двух групп посредством метода `counting()`:

```
Map<Boolean, Long> byWeightAndCount = melons.stream()
    .collect(partitioningBy(m -> m.getWeight() > 2000, counting()));
```

Результат будет таким:

```
{false=4, true=5}
```

Мы также можем подсчитать число элементов без повторов:

```
Map<Boolean, Long> byWeight = melons.stream()
    .distinct()
    .collect(partitioningBy(m -> m.getWeight() > 2000, counting()));
```

На этот раз результат будет таким:

```
{false=4, true=3}
```

Наконец, коллектор `partitioningBy()` может быть совмещен с методом `collectingAndThen()`, с которым мы познакомились в *разд. 189 "Группирование"*. Например, давайте разделим дыни по весу 2000 г и оставим только самые тяжелые из каждого подраздела:

```
Map<Boolean, Melon> byWeightMax = melons.stream()
    .collect(partitioningBy(m -> m.getWeight() > 2000,
        collectingAndThen(maxBy(comparingInt(Melon::getWeight)), Optional::get)));
```

Результат будет таким:

```
{false=Horned(1700g), true=Gac(3000g)}
```

191. Фильтрующий, сглаживающий и отображающий коллекторы

Допустим, что у нас есть класс `Melon` и список его объектов:

```
public class Melon {
    private final String type;
    private final int weight;
    private final List<String> pests;

    // конструкторы, геттеры, сеттеры, equals(),
    // hashCode(), toString() опущены для краткости
}

List<Melon> melons = Arrays.asList(new Melon("Crenshaw", 2000),
    new Melon("Hemi", 1600), new Melon("Gac", 3000),
    new Melon("Hemi", 2000), new Melon("Crenshaw", 1700),
    new Melon("Gac", 3000), new Melon("Hemi", 2600));
```

API потоков Java предусматривает коллекторы `filtering()`, `flatMaping()` и `mapping()`, специально для использования в многоуровневых редукциях (например, ниже в потоке после коллекторов `groupingBy()` или `partitioningBy()`).

Концептуально цель коллектора `filtering()` является такой же, как и у метода `filter()`, цель коллектора `flatMaping()` аналогична цели метода `flatMap()`, а цель коллектора `mapping()` совпадает с целью метода `map()`.

Коллектор *filtering()*

Задача пользователя: требуется взять все дыни, которые тяжелее 2000 г, и сгруппировать их по сорту. Добавить каждый сорт в надлежащий контейнер (у каждого сорта есть свой контейнер — просто проверить метки контейнера).

С помощью коллектора `filtering(Predicate<? super T> predicate, Collector<? Super T,A,R> downstream)` мы применяем предикат к каждому элементу текущего коллектора и накапливаем результат в нижестоящем коллекторе.

Таким образом, для того чтобы сгруппировать дыни, которые тяжелее 2000 г, по сорту, мы можем написать следующий потоковый конвейер:

```
Map<String, Set<Melon>> melonsFiltering = melons.stream()
    .collect(groupingBy(Melon::getType,
        filtering(m -> m.getWeight() > 2000, toSet())));
```

Результат будет таким (каждое множество `Set<Melon>` представляет собой контейнер):

```
{Crenshaw=[], Gac=[Gac(3000g)], Hemi=[Hemi(2600g)]}
```

Обратите внимание, что нет дыни Crenshaw тяжелее 2000 г, поэтому коллектор `filtering()` отобразил этот тип в пустое множество (контейнер). Теперь перепишем это с использованием метода `filter()`:

```
Map<String, Set<Melon>> melonsFiltering = melons.stream()
    .filter(m -> m.getWeight() > 2000)
    .collect(groupingBy(Melon::getType, toSet()));
```

Поскольку метод `filter()` не выполняет попарных сопоставлений для элементов, которые делают предикат равным `false`, то результат будет выглядеть следующим образом:

```
{Gac=[Gac(3000g)], Hemi=[Hemi(2600g)]}
```

Задача пользователя: на этот раз отыскать только дыни сорта дыни Hemi. Есть два контейнера: один для дыни Hemi легче (или равной) 2000 г и один для дыни Hemi тяжелее 2000 г.

Фильтрация может также использоваться с коллектором `partitioningBy()`. Для того чтобы разбить список с дынями тяжелее 2000 г и отфильтровать их по некоторому сорту (в данном случае, сорту дыни Hemi), воспользуемся таким кодом:

```
Map<Boolean, Set<Melon>> melonsFiltering = melons.stream()
    .collect(partitioningBy(m -> m.getWeight() > 2000,
        filtering(m -> m.getType().equals("Hemi"), toSet())));
```

Результат выглядит следующим образом:

```
{false=[Hemi(1600g), Hemi(2000g)], true=[Hemi(2600g)]}
```

Применение метода `filter()` приведет к тому же результату:

```
Map<Boolean, Set<Melon>> melonsFiltering = melons.stream()
    .filter(m -> m.getType().equals("Hemi"))
    .collect(partitioningBy(m -> m.getWeight() > 2000, toSet()));
```

Результат:

```
{false=[Hemi(1600g), Hemi(2000g)], true=[Hemi(2600g)]}
```

Коллектор *mapping()*

Задача пользователя: для каждого сорта дыни требуется получить список весов в возрастающем порядке.

С помощью коллектора `mapping(Function<? super T,? extends U> mapper, Collector<? super U,A,R> downstream)` мы можем применить функцию отображения к каждому элементу текущего коллектора и накопить результат в нижестоящем коллекторе.

Например, для группирования весов дынь по сорту можно написать такой фрагмент кода:

```
Map<String, TreeSet<Integer>> melonsMapping = melons.stream()
    .collect(groupingBy(Melon::getType,
        mapping(Melon::getWeight, toCollection(TreeSet::new))));
```

Результат:

```
{Crenshaw=[1700, 2000], Gac=[3000], Hemi=[1600, 2000, 2600]}
```

Задача пользователя: требуется получить два списка. Один из них должен содержать сорта дынь, которые легче (или равны) 2000 г, а другой — остальные сорта.

Выбрать дыни, которые тяжелее 2000 г, и собрать только их сорта можно следующим образом:

```
Map<Boolean, Set<String>> melonsMapping = melons.stream()
    .collect(partitioningBy(m -> m.getWeight() > 2000,
        mapping(Melon::getType, toSet())));
```

Результат выглядит так:

```
{false=[Crenshaw, Hemi], true=[Gac, Hemi]}
```

Коллектор *flatMap()*

В качестве быстрого напоминания о сглаживании потока рекомендуется прочитать *разд. 182 "Отображение элементов потока в новый поток"* ранее в этой главе.

Теперь допустим, что у нас есть следующий ниже список объектов класса `Melon` (обратите внимание, что мы также добавили имена вредителей):

```
List<Melon> melonsGrown = Arrays.asList(
    new Melon("Honeydew", 5600,
        Arrays.asList("Spider Mites", "Melon Aphids", "Squash Bugs")),
    new Melon("Crenshaw", 2000,
        Arrays.asList("Pickleworms")),
    new Melon("Crenshaw", 1000,
        Arrays.asList("Cucumber Beetles", "Melon Aphids")),
    new Melon("Gac", 4000,
        Arrays.asList("Spider Mites", "Cucumber Beetles")),
    new Melon("Gac", 1000,
        Arrays.asList("Squash Bugs", "Squash Vine Borers")));
```

Задача пользователя: для каждого сорта дыни сформировать список его вредителей.

Итак, давайте сгруппируем дыни по сортам и соберем их вредителей. Каждая дыня имеет ноль, одного или нескольких вредителей, и поэтому мы ожидаем результат в виде отображения `Map<String, List<String>>`. Первая попытка будет опираться на метод `mapping()`:

```
Map<String, List<List<String>>> pests = melonsGrown.stream()
    .collect(groupingBy(Melon::getType,
        mapping(m -> m.getPests(), toList())));
```

Очевидно, что этот подход не очень хороший, т. к. возвращаемым типом будет `Map<String, List<List<String>>>`.

Еще один наивный подход, основанный на отображении, заключается в следующем:

```
Map<String, List<List<String>>> pests = melonsGrown.stream()
    .collect(groupingBy(Melon::getType,
        mapping(m -> m.getPests().stream(), toList())));
```

Очевидно, что этот подход тоже не очень удачный, т. к. возвращаемым типом будет `Map<String, List<Stream<String>>>`.

Самое время ввести коллектор `flatMap()`. Используя коллектор `flatMap(Function<? Super T, ? extends Stream<? extends U>> mapper, Collector<? super U, A, R> downstream)`, мы применяем его к каждому элементу текущего коллектора и накапливаем результат в нижестоящем коллекторе:

```
Map<String, Set<String>> pestsFlatMapping = melonsGrown.stream()
    .collect(groupingBy(Melon::getType,
        flatMapping(m -> m.getPests().stream(), toSet())));
```

На этот раз тип выглядит нормально, и на выходе мы получаем вот такой результат:

```
{
    Crenshaw = [Cucumber Beetles, Pickleworms, Melon Aphids],
    Gac = [Cucumber Beetles, Squash Bugs, Spider Mites, Squash Vine Borers],
    Honeydew = [Squash Bugs, Spider Mites, Melon Aphids]
}
```

Задача пользователя: требуется получить два списка. Один должен содержать вредителей дынь легче 2000 г, а другой — вредителей остальных дынь.

Выделить список дынь, которые тяжелее 2000 г, и собрать вредителей можно так:

```
Map<Boolean, Set<String>> pestsFlatMapping = melonsGrown.stream()
    .collect(partitioningBy(m -> m.getWeight() > 2000,
        flatMapping(m -> m.getPests().stream(), toSet())));
```

Результат выглядит следующим образом:

```
{
    false = [Cucumber Beetles, Squash Bugs, Pickleworms, Melon Aphids,
        Squash Vine Borers],
    true = [Squash Bugs, Cucumber Beetles, Spider Mites, Melon Aphids]
}
```


192. Сочленение

Начиная с JDK 12, мы можем объединить результаты двух коллекторов посредством коллектора `Collectors.teeing()`:

```
public static <T,R1,R2,R> Collector<T,?,R> teeing(Collector<? super T,?,R1>
downstream1, Collector<? super T,?,R2> downstream2, BiFunction<? super R1,? super
R2,R> merger)
```

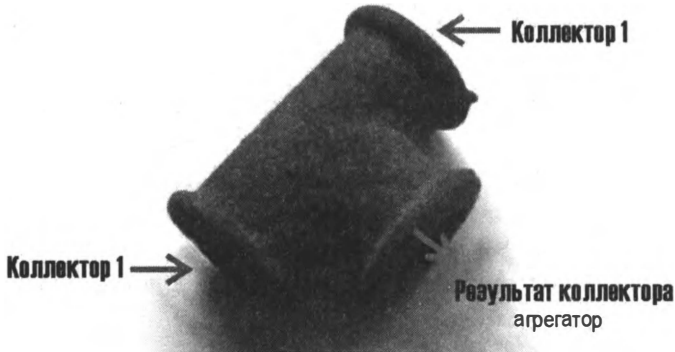


Рис. 9.15

В результате получается коллектор, представляющий собой композит из двух прошедших по потоку коллекторов (рис. 9.15). Каждый элемент, прошедший в результирующий коллектор, обрабатывается обоими нижестоящими коллекторами, а затем их результаты соединяются в финальный результат с помощью заданной би-функции `BiFunction`.

Давайте взглянем на классическую задачу. Следующий ниже класс хранит число элементов в целочисленном потоке и их сумму:

```
public class CountSum {
    private final Long count;
    private final Integer sum;

    public CountSum(Long count, Integer sum) {
        this.count = count;
        this.sum = sum;
    }
    ...
}
```

Мы можем получить эту информацию посредством коллектора `teeing()` следующим образом:

```
CountSum countsum = Stream.of(2, 11, 1, 5, 7, 8, 12)
    .collect(Collectors.teeing(
        counting(),
        summingInt(e -> e),
        CountSum::new));
```

Здесь мы применили два коллектора к каждому элементу из потока (`counting()` и `summingInt()`), и результаты были объединены в экземпляре класса `CountSum`:

```
CountSum{count=7, sum=46}
```

Давайте рассмотрим еще одну задачу. На этот раз класс `MinMax` хранит минимум и максимум целочисленного потока:

```
public class MinMax {
    private final Integer min;
    private final Integer max;

    public MinMax(Integer min, Integer max) {
        this.min = min;
        this.max = max;
    }
    ...
}
```

Теперь мы можем получить эту информацию следующим образом:

```
MinMax minmax = Stream.of(2, 11, 1, 5, 7, 8, 12)
    .collect(Collectors.teeing(
        minBy(Comparator.naturalOrder()),
        maxBy(Comparator.naturalOrder()),
        (Optional<Integer> a, Optional<Integer> b)
        -> new MinMax(a.orElse(Integer.MIN_VALUE),
            b.orElse(Integer.MAX_VALUE))));
```

Здесь мы применили два коллектора к каждому элементу из потока (`minBy()` и `maxBy()`), и результаты были объединены в экземпляре класса `MinMax`:

```
MinMax{min=1, max=12}
```

Наконец, давайте рассмотрим класс `Melon` и список его объектов:

```
public class Melon {
    private final String type;
    private final int weight;

    public Melon(String type, int weight) {
        this.type = type;
        this.weight = weight;
    }
    ...
}
```

```
List<Melon> melons = Arrays.asList(new Melon("Crenshaw", 1200),
    new Melon("Gac", 3000), new Melon("Hemi", 2600),
    new Melon("Hemi", 1600), new Melon("Gac", 1200),
    new Melon("Apollo", 2600), new Melon("Horned", 1700),
    new Melon("Gac", 3000), new Melon("Hemi", 2600));
```

Цель здесь — вычислить суммарный вес этих дынь и вывести список их весов. Мы можем показать это следующим образом:

```
public class WeightsAndTotal {
    private final int totalWeight;
    private final List<Integer> weights;

    public WeightsAndTotal(int totalWeight, List<Integer> weights) {
        this.totalWeight = totalWeight;
        this.weights = weights;
    }
    ...
}
```

Решение этой задачи опирается на коллектор `Collectors.teeing()`:

```
WeightsAndTotal weightsAndTotal = melons.stream()
    .collect(Collectors.teeing(
        summingInt(Melon::getWeight),
        mapping(m -> m.getWeight(), toList()),
        WeightsAndTotal::new));
```

На этот раз мы применили коллекторы `summingInt()` и `mapping()`. Результат выглядит следующим образом:

```
WeightsAndTotal {
    totalWeight = 19500,
    weights = [1200, 3000, 2600, 1600, 1200, 2600, 1700, 3000, 2600]
}
```

193. Написание собственного коллектора

Допустим, что у нас есть класс `Melon` и список его объектов:

```
public class Melon {
    private final String type;
    private final int weight;
    private final List<String> grown;

    // конструкторы, геттеры, сеттеры, equals(),
    // hashCode(), toString() опущены для краткости
}
```

```
List<Melon> melons = Arrays.asList(new Melon("Crenshaw", 1200),
    new Melon("Gac", 3000), new Melon("Hemi", 2600),
    new Melon("Hemi", 1600), new Melon("Gac", 1200),
    new Melon("Apollo", 2600), new Melon("Horned", 1700),
    new Melon("Gac", 3000), new Melon("Hemi", 2600));
```

В разд. 191 "Разбиение" мы видели, как использовать коллектор `partitioningBy()` для выделения дынь весом 2000 г с повторами:

```
Map<Boolean, List<Melon>> byWeight = melons.stream()
    .collect(partitioningBy(m -> m.getWeight() > 2000));
```

Теперь давайте посмотрим, сможем ли мы достичь того же результата с помощью собственного специализированного коллектора.

Начнем с того, что признаем: операция написания собственного коллектора не является повседневной, но знание о том, как ее выполнять правильно, может оказаться полезным. Встроенный в Java интерфейс `Collector` выглядит следующим образом:

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
    Set<Characteristics> characteristics();
    ...
}
```

Для того чтобы написать собственный коллектор, очень важно знать, что типы `T`, `A` и `R` представляют собой следующее:

- ◆ `T` представляет тип элементов из потока `Stream` (элементы, которые будут собраны);
- ◆ `A` представляет тип объекта, использованного во время процесса сбора и именуемого *накопителем*, который применяется для накопления элементов потока в *контейнере мутируемых результатов*;
- ◆ `R` представляет тип объекта после процесса сбора (финальный результат).

Коллектор может вернуть сам накопитель в качестве финального результата или выполнить дополнительное преобразование на накопителе для получения финального результата (выполнить дополнительное финальное преобразование из типа промежуточного накопления `A` в тип финального результата `R`).

В терминах нашей задачи мы знаем, что `T` — это объект `Melon`, `A` — отображение `Map<Boolean, List<Melon>>`, `R` — отображение `Map<Boolean, List<Melon>>`. Этот коллектор возвращает сам накопитель в качестве финального результата посредством метода `Function.identity()`. С учетом сказанного мы можем записать начальные строки кода нашего собственного коллектора следующим образом:

```
public class MelonCollector implements
    Collector<Melon, Map<Boolean, List<Melon>>, Map<Boolean, List<Melon>>> {
    ...
}
```

Таким образом, коллектор задается четырьмя функциями. Эти функции работают вместе, чтобы накапливать элементы в контейнере мутируемых результатов и опционально выполнять финальное преобразование результата.

Они заключаются в следующем:

- ◆ создание нового пустого контейнера мутируемых результатов (`supplier()`);
- ◆ встраивание нового элемента данных в контейнер мутируемых результатов (`accumulator()`);
- ◆ объединение двух контейнеров мутируемых результатов в один (`combiner()`);
- ◆ выполнение необязательного финального преобразования в контейнере мутируемых результатов для получения финального результата (`finisher()`).

В дополнение к этому, поведение коллектора определяется в последнем методе, `characteristics()`. Множество `Set<Characteristics>` может содержать следующие четыре значения:

- ◆ `UNORDERED` — порядок накопления/сбора элементов не важен для финального результата;
- ◆ `CONCURRENT` — элементы потока могут накапливаться несколькими нитями исполнения в конкурентном стиле (в конце коллектор может выполнять параллельную редукцию потока. Контейнеры, получаемые в результате параллельной обработки потока, объединяются в один результирующий контейнер. Источник данных должен быть неупорядоченным по своей природе или должен присутствовать флаг `UNORDERED`;
- ◆ `IDENTITY_FINISH` — указывает на то, что сам накопитель является финальным результатом (в сущности, мы можем явно привести тип `A` к типу `R`); в этом случае `finisher()` не вызывается.

Поставщик ***Supplier<A> supplier()***;

Работа метода `supplier()` — возвращать (при каждом вызове) поставщика `Supplier` пустого контейнер мутируемых результатов.

В нашем случае результирующий контейнер имеет тип `Map<Boolean, List<Melon>>`, и поэтому метод `supplier()` может быть имплементирован следующим образом:

```
@Override
public Supplier<Map<Boolean, List<Melon>>> supplier() {
    return () -> {
        return new HashMap<Boolean, List<Melon>> () {
            {
                put(true, new ArrayList<>());
                put(false, new ArrayList<>());
            }
        };
    };
}
```

При параллельном исполнении этот метод может вызываться несколько раз.

Накопление элементов — *BiConsumer<A, T> accumulator()*;

Метод `accumulator()` возвращает функцию, которая выполняет операцию редукции. Этой функцией является `BiConsumer` — операция, которая принимает два входных аргумента и не возвращает результата. Первый входной аргумент — это контейнер текущих результатов (являющийся результатом редукции до этого момента), второй входной аргумент — это текущий элемент из потока. Указанная функция модифицирует сам контейнер результатов, накапливая пройденный элемент или эффект обхода этого элемента. В нашем случае `accumulator()` добавляет текущий пройденный элемент в один из двух списков `ArrayList`:

```
@Override
public BiConsumer<Map<Boolean, List<Melon>>, Melon> accumulator() {

    return (var acc, var melon) -> {
        acc.get(melon.getWeight() > 2000).add(melon);
    };
}
```

Применение финального преобразования — *Function<A, R> finisher()*;

Метод `finisher()` возвращает функцию, которая применяется в конце процесса накопления. Когда этот метод вызывается, в потоке больше нет элементов, подлежащих обходу. Все элементы будут накапливаться преобразованными из типа промежуточного накопления `A` в тип финального результата `R`. Если преобразование не требуется, мы можем вернуть промежуточный результат (сам накопитель):

```
@Override
public Function<Map<Boolean, List<Melon>>,
    Map<Boolean, List<Melon>>> finisher() {

    return Function.identity();
}
```

Параллелизация коллектора — *BinaryOperator<A> combiner()*;

Если поток обрабатывается параллельно, то разные нити исполнения (накопители) будут производить контейнеры частичных результатов. В конце эти частичные результаты должны быть объединены в один. Это именно то, что делает `combiner()`. В данном случае метод `combiner()` должен объединить два отображения, добавив все значения из двух списков второго отображения в соответствующие списки первого отображения:

```
@Override
public BinaryOperator<Map<Boolean, List<Melon>>> combiner() {
```

```
return (var map, var addMap) -> {
    map.get(true).addAll(addMap.get(true));
    map.get(false).addAll(addMap.get(false));

    return map;
};
}
```

Возврат финального результата — *Function<A, R> finisher();*

Финальный результат вычисляется в методе `finisher()`. В данном случае мы просто возвращаем `Function.identity()`, т. к. накопитель не требует дальнейшего преобразования:

```
@Override
public Function<Map<Boolean, List<Melon>>,
    Map<Boolean, List<Melon>>> finisher() {

    return Function.identity();
}
```

Характеристики — *Set<Characteristics> characteristics();*

Наконец, мы указываем, что наш коллектор является `IDENTITY_FINISH` и `CONCURRENT`:

```
@Override
public Set<Characteristics> characteristics() {
    return Set.of(IDENTITY_FINISH, CONCURRENT);
}
```

Исходный код, который прилагается к этой книге, склеивает все части головоломки вместе в классе под названием `MelonCollector`.

Проверка времени

Класс `MelonCollector` может использоваться с ключевым словом `new` так:

```
Map<Boolean, List<Melon>> melons2000 = melons.stream()
    .collect(new MelonCollector());
```

Мы получим следующий результат:

```
{
    false = [Crenshaw(1200 g),Hemi(1600 g),Gac(1200 g),Horned(1700 g)],
    true = [Gac(3000 g),Hemi(2600 g),Apollo(2600 g),
        Gac(3000 g),Hemi(2600 g)]
}
```

Мы также можем использовать его посредством метода `parallelStream()`:

```
Map<Boolean, List<Melon>> melons2000 = melons.parallelStream()
    .collect(new MelonCollector());
```

Если мы применяем метод `combiner()`, то результат может выглядеть следующим образом:

```
{false = [], true = [Hemi(2600g)]}
    ForkJoinPool.commonPool - worker - 7
...
{false = [Horned(1700g)], true = []}
    ForkJoinPool.commonPool - worker - 15
{false = [Crenshaw(1200g)], true = [Gac(3000g)]}
    ForkJoinPool.commonPool - worker - 9
...
{false = [Crenshaw(1200g), Hemi(1600g), Gac(1200g), Horned(1700g)],
true = [Gac(3000g), Hemi(2600g), Apollo(2600g),
    Gac(3000g), Hemi(2600g)]}
```

Собственный коллектор посредством метода `collect()`

В случае операции сбора `IDENTITY_FINISH` существует, по крайней мере, еще один вариант решения задачи получения собственного коллектора. Этому варианту решения способствует следующий ниже метод:

```
<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T>
accumulator, BiConsumer<R,R> combiner)
```

Данная разновидность метода `collect()` отлично подходит в ситуациях, когда мы имеем дело с операцией сбора `IDENTITY_FINISH`, и мы можем предоставить поставщика, накопитель и объединитель.

Давайте рассмотрим несколько примеров:

```
List<String> numbersList = Stream.of("One", "Two", "Three")
    .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);

Deque<String> numbersDeque = Stream.of("One", "Two", "Three")
    .collect(ArrayDeque::new, ArrayDeque::add, ArrayDeque::addAll);

String numbersString = Stream.of("One", "Two", "Three")
    .collect(StringBuilder::new, StringBuilder::append,
        StringBuilder::append).toString();
```

Вы можете взять на вооружение эти примеры для выявления большего числа классов JDK, сигнатуры которых хорошо подходят для использования со ссылками на методы в качестве аргументов метода `collect()`.

194. Ссылка на метод

Допустим, что у нас есть класс `Melon` и список объектов этого класса:

```
public class Melon {
    private final String type;
    private int weight;
```



```
public static int growing100g(Melon melon) {
    melon.setWeight(melon.getWeight() + 100);
    return melon.getWeight();
}

// конструкторы, геттеры, сеттеры, equals(),
// hashCode(), toString() omitted for brevity
}

List<Melon> melons = Arrays.asList(
    new Melon("Crenshaw", 1200), new Melon("Gac", 3000),
    new Melon("Hemi", 2600), new Melon("Hemi", 1600));
```

Ссылки на методы являются сокращениями для лямбда-выражений.

В общих чертах ссылка на метод представляет собой технический прием, который используется для вызова метода по имени, а не по описанию того, как его вызывать. Главное ее преимущество — удобочитаемость.

Ссылка на метод записывается путем размещения целевой ссылки перед разделителем `::`, а имя метода указывается после него.

Мы рассмотрим все четыре вида ссылок на методы в следующих далее разделах.

Ссылка на статический метод

Мы можем сгруппировать каждую дыню из вышеупомянутого списка, которая весит 100 г, посредством статического метода `growing100g()`:

- ◆ ссылка на метод отсутствует:
`melons.forEach(m -> Melon.growing100g(m));`
- ◆ ссылка на метод:
`melons.forEach(Melon::growing100g);`

Ссылка на экземплярный метод

Допустим, что мы определяем следующий ниже компаратор для класса `Melon`:

```
public class MelonComparator implements Comparator {
    @Override
    public int compare(Object m1, Object m2) {
        return Integer.compare(((Melon) m1).getWeight(),
            ((Melon) m2).getWeight());
    }
}
```

Теперь мы можем сослаться на него следующим образом:

◆ ссылка на метод отсутствует:

```
MelonComparator mc = new MelonComparator();

List<Melon> sorted = melons.stream()
    .sorted((Melon m1, Melon m2) -> mc.compare(m1, m2))
    .collect(Collectors.toList());
```

◆ ссылка на метод:

```
List<Melon> sorted = melons.stream()
    .sorted(mc::compare)
    .collect(Collectors.toList());
```

Конечно, мы также можем вызвать метод `Integer.compare()` непосредственно:

◆ ссылка на метод отсутствует:

```
List<Integer> sorted = melons.stream()
    .map(m -> m.getWeight())
    .sorted((m1, m2) -> Integer.compare(m1, m2))
    .collect(Collectors.toList());
```

◆ ссылка на метод:

```
List<Integer> sorted = melons.stream()
    .map(m -> m.getWeight())
    .sorted(Integer::compare)
    .collect(Collectors.toList());
```

Ссылка на конструктор

Обращение по ссылке к конструктору выполняется посредством ключевого слова `new` следующим образом:

```
BiFunction<String, Integer, Melon> melonFactory = Melon::new;
Melon hemil300 = melonFactory.apply("Hemi", 1300);
```

Дополнительные сведения и примеры о ссылке на конструктор доступны в *разд. 169 "Имплементирование шаблона фабрики" главы 8*.

195. Параллельная обработка потоков

Параллельная обработка потока относится к процессу, который состоит из трех шагов:

1. Разбиение элементов потока на несколько фрагментов.
2. Обработка каждого фрагмента в отдельной нити исполнения.
3. Объединение результатов обработки в общий результат.

Эти три шага происходят за кулисами посредством метода по умолчанию класса `ForkJoinPool`, как мы обсуждаем в *главе 10*, а также в *главе 11*.

В качестве общего правила следует запомнить, что параллельная обработка может применяться только к *операциям без состояния* (состояние элемента не влияет на другой элемент), *невмешивающимся операциям* (источник данных не подвергается влиянию) и *ассоциативным операциям* (результат не подвергается влиянию со стороны порядка следования операндов).

Допустим, что наша задача состоит в суммировании элементов списка значений типа `double`:

```
Random rnd = new Random();
List<Double> numbers = new ArrayList<>();

for (int i = 0; i < 1_000_000; i++) {
    numbers.add(rnd.nextDouble());
}
```

Мы также можем сделать это непосредственно в качестве потока:

```
DoubleStream.generate(() -> rnd.nextDouble()).limit(1_000_000)
```

При последовательном подходе мы можем сделать это следующим образом:

```
double result = numbers.stream()
    .reduce((a, b) -> a + b).orElse(-1d);
```

Эта операция за кулисами, вероятно, будет происходить на одном ядре (даже если наша машина имеет больше ядер), как показано на рис. 9.16.



Рис. 9.16

Для решения этой проблемы будет разумным привлечение параллелизации, и поэтому мы можем вызвать метод `parallelStream()` вместо метода `stream()`, как показано ниже:

```
double result = numbers.parallelStream()
    .reduce((a, b) -> a + b).orElse(-1d);
```

После того как мы вызовем `parallelStream()`, Java начнет действовать и обработает поток, используя несколько нитей исполнения. Параллелизация также может быть выполнена посредством метода `parallel()`:

```
double result = numbers.stream()
    .parallel()
    .reduce((a, b) -> a + b).orElse(-1d);
```

На этот раз обработка происходит посредством разветвления/соединения, как показано на рис. 9.17 (для каждого доступного ядра имеется одна нить исполнения).

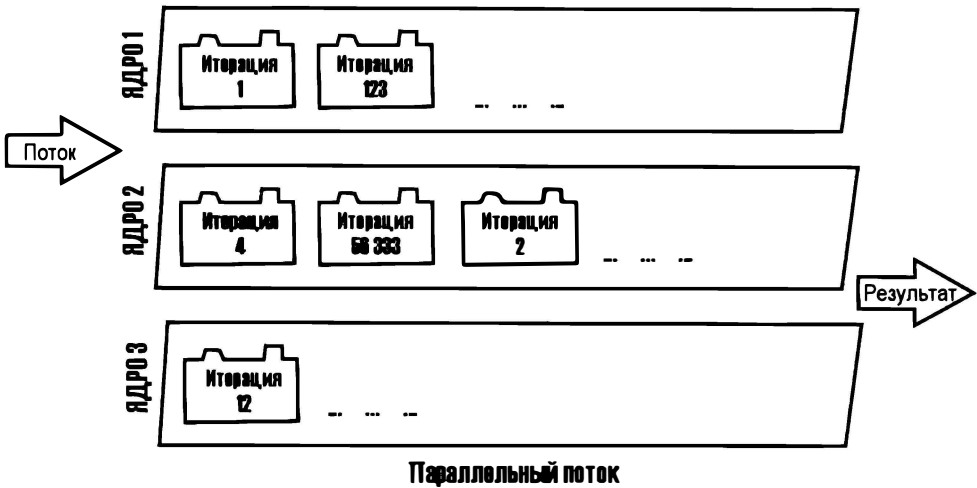


Рис. 9.17

В контексте метода `reduce()` параллелизация может быть изображена так, как показано на рис. 9.18.

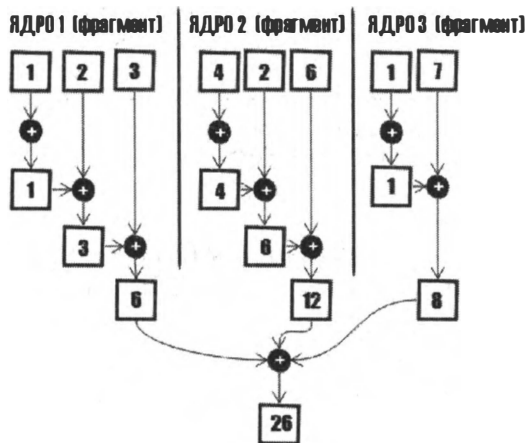


Рис. 9.18

По умолчанию класс `ForkJoinPool` среды Java попытается получить столько нитей исполнения, сколько доступно процессоров:

```
int noOfProcessors = Runtime.getRuntime().availableProcessors();
```

Мы можем повлиять на число нитей исполнения глобально (все параллельные потоки будут его использовать) следующим образом:

```
System.setProperty(
    "java.util.concurrent.ForkJoinPool.common.parallelism", "10");
```

Мы также можем повлиять на число нитей исполнения для отдельного параллельного потока следующим образом:

```
ForkJoinPool customThreadPool = new ForkJoinPool(5);
```

```
double result = customThreadPool.submit(
    () -> numbers.parallelStream()
        .reduce((a, b) -> a + b)).get().orElse(-1d);
```

Важно принять решение о влиянии на число нитей исполнения. Определить оптимальное число нитей исполнения в зависимости от среды не просто, и в большинстве сценариев наиболее подходящей является стандартная конфигурация (*число нитей исполнения = число процессоров*).



Даже если задача — хороший кандидат для задействования параллелизации, это не означает, что параллельная обработка является волшебной пилюлей. Решение о переходе на параллельную обработку должно приниматься после сравнительного анализа последовательной и параллельной обработки. Чаще всего параллельная обработка показывает более высокую производительность в случае огромных наборов данных.

Не льстите себя надеждой, думая, что большее число нитей исполнения приводит к более быстрой обработке. Избегайте чего-то вроде следующего (эти показатели являются всего лишь индикаторами для машины с 8 ядрами):

```
5 нитей исполнения (~40 ms)
20 нитей исполнения (~50 ms)
100 нитей исполнения (~70 ms)
1000 нитей исполнения (~ 250 ms)
```

Сплитераторы

Интерфейс `Splititerator` среды Java (от англ. *splittable iterator* — разбиваемый итератор) — это интерфейс, который используется для параллельного обхода элементов источника (например, коллекции или потока). Этот интерфейс определяет следующие методы:

```
public interface Splititerator<T> {
    boolean tryAdvance(Consumer<? super T> action);
    Splititerator<T> trySplit();
    long estimateSize();
    int characteristics();
}
```

Рассмотрим простой список из 10 целых чисел:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

Мы можем получить `Splititerator` для этого списка следующим образом:

```
Splititerator<Integer> s1 = numbers.splititerator();
```

Мы также можем сделать то же самое из потока:

```
Splititerator<Integer> s1 = numbers.stream().splititerator();
```

Для того чтобы перейти к первому элементу, нам нужно вызвать метод `tryAdvance()`, как показано ниже:

```
s1.tryAdvance(e -> System.out.println("Переход к первому элементу s1: " + e));
```

Мы получим вот такой результат:

Переход к первому элементу s1: 1

Сплитератор может попытаться оценить число элементов, оставшихся для обхода посредством метода `estimateSize()`:

```
System.out.println("\nРасчетный размер s1: " + s1.estimateSize());
```

Мы получим следующий результат (мы прошли один элемент; осталось пройти девять):

Расчетный размер s1: 9

Мы можем разбить его на два сплитератора, используя метод `trySplit()`. Результатом будет еще один объект `Spliterator`:

```
Spliterator<Integer> s2 = s1.trySplit();
```

Проверка числа элементов выявляет воздействие метода `trySplit()`:

```
System.out.println("Расчетный размер s1: " + s1.estimateSize());
```

```
System.out.println("Расчетный размер s2: " + s2.estimateSize());
```

Мы получим следующий результат:

Расчетный размер s1: 5

Расчетный размер s2: 4

Попытаться вывести все элементы из `s1` и `s2` можно с помощью функции `forEachRemaining()` следующим образом:

```
s1.forEachRemaining(System.out::println); // 6, 7, 8, 9, 10
```

```
s2.forEachRemaining(System.out::println); // 2, 3, 4, 5
```

Интерфейс `Spliterator` определяет связку констант для его характеристики — `CONCURRENT (4096)`, `DISTINCT (1)`, `IMMUTABLE (1024)`, `NONNULL (256)`, `ORDERED (16)`, `SIZED (64)`, `SORTED (4)` и `SUBSIZED (16 384)`.

Мы можем вывести характеристики с помощью метода `characteristics()` следующим образом:

```
System.out.println(s1.characteristics()); // 16464
```

```
System.out.println(s2.characteristics()); // 16464
```

Проще проверить, что некоторая характеристика представлена, можно с помощью метода `hasCharacteristics()`:

```
if (s1.hasCharacteristics(Spliterator.ORDERED)) {  
    System.out.println("ORDERED");  
}
```

```
if (s1.hasCharacteristics(Spliterator.SIZED)) {  
    System.out.println("SIZED");  
}
```

Написание собственного сплитератора

Очевидно, что задача написания собственного сплитератора не является ежедневной, но давайте допустим, что мы работаем над проектом, который по какой-то причине требует от нас обработки строк, содержащих идеографические символы (Chinese, Japanese, Korean и Vietnamese, CJKV — символы китайского, японского, корейского и вьетнамского языков) и неидеографические символы. Мы хотим обрабатывать эти строки в параллельном режиме. Это требует, чтобы мы разбили их на символы только в позициях, представляющих идеографические символы.

Очевидно, что `Spliterator` не будет работать так, как мы хотим, и поэтому нам может потребоваться написать собственный сплитератор. Для этого мы должны имплементировать интерфейс `Spliterator` и обеспечить имплементацию нескольких методов. Его имплементация имеется в исходном коде, который прилагается к этой книге. Подумайте о том, чтобы открыть исходный код интерфейса `IdeographicSpliterator` и держать его рядом во время чтения остальной части этого раздела.

Кульминацией его имплементации является метод `trySplit()`. Здесь мы пытаемся разделить текущее строковое значение пополам и продолжать ее обход до тех пор, пока не найдем идеографический символ. Для целей проверки мы только что добавили следующую ниже строку кода:

```
System.out.println("Разбито успешно на символе: "
    + str.charAt(splitPosition));
```

Теперь рассмотрим строковое значение, содержащее идеографические символы:

```
String str = "Character Information 字 Development and Maintenance "
    + "Project 盤 for e-Government MojiJoho-Kiban 事 Project";
```

Затем создадим параллельный поток для этого строкового значения и заставим `IdeographicSpliterator` сделать свое дело:

```
Spliterator<Character> spliterator = new IdeographicSpliterator(str);
Stream<Character> stream = StreamSupport.stream(spliterator, true);
```

```
// заставить spliterator сделать свою работу
stream.collect(Collectors.toList());
```

Один из возможных результатов покажет, что разбиение происходит только в позициях, содержащих идеографические символы:

Успешно разбито на символе: 盤

Успешно разбито на символе: 事

196. Null-безопасные потоки

Проблема с созданием потока из элемента, который может быть равен `null`, может быть решена с помощью метода `Optional.ofNullable()` или, еще лучше, посредством метода `Stream.ofNullable()` JDK 9:

```
static <T> Stream<T> ofNullable(T t)
```

Этот метод получает один-единственный элемент (T) и возвращает последовательный поток, содержащий этот единственный элемент (Stream<T>); в противном случае метод возвращает пустой поток, если элемент равен null.

Например, мы можем написать вспомогательный метод, который оборачивает вызов метода Stream.ofNullable():

```
public static <T> Stream<T> elementAsStream(T element) {
    return Stream.ofNullable(element);
}
```

Если этот метод располагается в служебном классе с именем AsStreams, мы можем выполнить несколько вызовов, как показано ниже:

```
// 0
System.out.println("Null-элемент: "
    + AsStreams.elementAsStream(null).count());

// 1
System.out.println("Не-null-элемент: "
    + AsStreams.elementAsStream("Hello world").count());
```

Обратите внимание, что при передаче null мы получаем пустой поток (метод count() возвращает 0)!

Если нашим элементом является коллекция, то все становится интереснее. Например, допустим, что у нас есть следующий ниже список (обратите внимание, что этот список содержит несколько значений null):

```
List<Integer> ints = Arrays.asList(5, null, 6, null, 1, 2);
```

Теперь давайте напишем вспомогательный метод, который возвращает Stream<T>, где T — это коллекция:

```
public static <T> Stream<T> collectionAsStreamWithNulls(
    Collection<T> element) {
    return Stream.ofNullable(element).flatMap(Collection::stream);
}
```

Если мы вызовем этот метод с null, то получим пустой поток:

```
// 0
System.out.println("Null-коллекция: "
    + AsStreams.collectionAsStreamWithNulls(null).count());
```

Теперь, если мы вызовем его с нашим списком ints, то получим Stream<Integer>:

```
// 6
System.out.println("Не-null-коллекция с элементами null: "
    + AsStreams.collectionAsStreamWithNulls(ints).count());
```

Обратите внимание, что поток содержит шесть элементов (все элементы из опорного списка) — 5, null, 6, null, 1 и 2.

Если мы знаем, что сама коллекция не является `null`, но может содержать значения `null`, то мы можем написать еще один вспомогательный метод:

```
public static <T> Stream<T> collectionAsStreamWithoutNulls(
    Collection<T> collection) {

    return collection.stream().flatMap(e -> Stream.ofNullable(e));
}
```

На этот раз, если сама коллекция равна `null`, то исходный код выбросит исключение `NullPointerException`. Однако если мы передадим ему наш список, то результатом будет `Stream<Integer>` без значений `null`:

```
// 4
System.out.println("Не-null-коллекция без элементов null: "
    + AsStreams.collectionAsStreamWithoutNulls(ints).count());
```

Возвращаемый поток содержит только четыре элемента — 5, 6, 1 и 2.

Наконец, если сама коллекция может быть `null` и содержать значения `null`, то следующий ниже помощник выполнит эту работу и вернет `null`-безопасный поток:

```
public static <T> Stream<T> collectionAsStream(
    Collection<T> collection) {

    return Stream.ofNullable(collection)
        .flatMap(Collection::stream)
        .flatMap(Stream::ofNullable);
}
```

Если мы передадим `null`, то получим пустой поток:

```
// 0
System.out.println("Null-коллекция или не-null-коллекция с элементами null: "
    + AsStreams.collectionAsStream(null).count());
```

Если мы передадим наш список, то получим поток `Stream<Integer>` без значений `null`:

```
// 4
System.out.println(
    "Null-коллекция или не-null-коллекция с элементами null: "
    + AsStreams.collectionAsStream(ints).count());
```

197. Композиция функций, предикатов и компараторов

Композиция (или выстраивание в цепочку) функций, предикатов и компараторов позволяет нам писать сложные критерии, которые должны применяться в унисон.

Композиция предикатов

Допустим, что у нас есть следующий класс `Melon` и список его объектов:

```
public class Melon {
    private final String type;
    private final int weight;

    // конструкторы, геттеры, сеттеры, equals(),
    // hashCode(), toString() опущены для краткости
}
```

```
List<Melon> melons = Arrays.asList(new Melon("Gac", 2000),
    new Melon("Horned", 1600), new Melon("Apollo", 3000),
    new Melon("Gac", 3000), new Melon("Hemi", 1600));
```

Интерфейс `Predicate` содержит три метода, которые берут `Predicate` и используют его для получения обогащенного предиката `Predicate`. Этими методами являются `and()`, `or()` и `negate()`.

Например, допустим, что мы хотим отфильтровать дыни, которые тяжелее 2000 г. Для этого мы можем написать предикат следующим образом:

```
Predicate<Melon> p2000 = m -> m.getWeight() > 2000;
```

Теперь мы хотим обогатить этот предикат, чтобы фильтровать только те дыни, которые соблюдают `p2000` и относятся к сорту дынь `Gac` или `Apollo`. Для этого мы можем использовать методы `and()` и `or()` следующим образом:

```
Predicate<Melon> p2000GacApollo
    = p2000.and(m -> m.getType().equals("Gac"))
    .or(m -> m.getType().equals("Apollo"));
```

Этот код интерпретируется слева направо как `a && (b || c)`, и мы имеем следующее:

- ◆ `a` is `m -> m.getWeight() > 2000;`
- ◆ `b` is `m -> m.getType().equals("Gac");`
- ◆ `c` is `m -> m.getType().equals("Apollo").`

Очевидно, что мы можем добавить больше критериев в том же стиле.

Давайте передадим этот предикат `Predicate` в метод `filter()`:

```
// Apollo(3000g), Gac(3000g)
List<Melon> result = melons.stream()
    .filter(p2000GacApollo)
    .collect(Collectors.toList());
```

Теперь допустим, что наша задача требует, чтобы мы получили отрицание вышеупомянутого составного предиката. Переписывать этот предикат как `!a && !b && !c` или любое другое аналогичное выражение будет громоздко. Оптимальным решением является вызов метода `negate()` следующим образом:

```
Predicate<Melon> restOf = p2000GacApollo.negate();
```

Давайте передадим его в метод `filter()`:

```
// Gac(2000g), Horned(1600g), Hemi(1600g)
List<Melon> result = melons.stream()
    .filter(restOf)
    .collect(Collectors.toList());
```

Начиная с JDK 11, мы можем отрицать предикат, который передается в качестве аргумента в метод `not()`. Например, давайте отфильтруем все дыни, которые легче (или равны) 2000 г, используя метод `not()`:

```
Predicate<Melon> pNot2000 = Predicate.not(m -> m.getWeight() > 2000);

// Gac(2000g), Horned(1600g), Hemi(1600g)
List<Melon> result = melons.stream()
    .filter(pNot2000)
    .collect(Collectors.toList());
```

Композиция компараторов

Рассмотрим тот же класс `Melon` и список его объектов из предыдущего раздела.

Теперь отсортируем этот список дынь по весу с помощью компаратора

```
Comparator.comparing():

Comparator<Melon> byWeight = Comparator.comparing(Melon::getWeight);

// Horned(1600g), Hemi(1600g), Gac(2000g), Apollo(3000g), Gac(3000g)
List<Melon> sortedMelons = melons.stream()
    .sorted(byWeight)
    .collect(Collectors.toList());
```

Мы также можем отсортировать список по сорту:

```
Comparator<Melon> byType = Comparator.comparing(Melon::getType);

// Apollo(3000g), Gac(2000g), Gac(3000g), Hemi(1600g), Horned(1600g)
List<Melon> sortedMelons = melons.stream()
    .sorted(byType)
    .collect(Collectors.toList());
```

Для того чтобы инвертировать порядок сортировки, следует просто вызвать метод `reversed()`:

```
Comparator<Melon> byWeight
    = Comparator.comparing(Melon::getWeight).reversed();
```

Пока все хорошо!

Теперь допустим, что мы хотим отсортировать список по весу и сорту. Другими словами, когда две дыни имеют одинаковый вес (например, `Horned(1600g)`, `Hemi(1600g)`), их следует отсортировать по сорту (например, `Hemi(1600g)`, `Horned(1600g)`).

Наивный подход будет выглядеть следующим образом:

```
// Apollo(3000g), Gac(2000g), Gac(3000g), Hemi(1600g), Horned(1600g)
List<Melon> sortedMelons = melons.stream()
    .sorted(byWeight)
    .sorted(byType)
    .collect(Collectors.toList());
```

Очевидно, что результат получится не такой, какой мы ожидали. Это происходит потому, что компараторы не были применены к одному и тому же списку. Компаратор `byWeight` применяется к оригинальному списку, тогда как компаратора `byType` применяется к результату компаратора `byWeight`. В сущности, компаратор `byType` отменяет эффект компаратора `byWeight`.

Решение приходит от метода `Comparator.thenComparing()`. Этот метод позволяет нам выстраивать компараторы в цепочку:

```
Comparator<Melon> byWeightAndType
    = Comparator.comparing(Melon::getWeight)
        .thenComparing(Melon::getType);

// Hemi(1600g), Horned(1600g), Gac(2000g), Apollo(3000g), Gac(3000g)
List<Melon> sortedMelons = melons.stream()
    .sorted(byWeightAndType)
    .collect(Collectors.toList());
```

Данная разновидность метода `thenComparing()` берет `Function` в качестве аргумента. Указанная функция используется для извлечения ключа сортировки `Comparable`. Возвращенный компаратор `Comparator` применяется только тогда, когда предыдущий компаратор нашел два эквивалентных объекта.

Еще одна разновидность метода `thenComparing()` получает компаратор:

```
Comparator<Melon> byWeightAndType =
    Comparator.comparing(Melon::getWeight)
        .thenComparing(Comparator.comparing(Melon::getType));
```

Напоследок рассмотрим следующий ниже список объектов класса `Melon`:

```
List<Melon> melons = Arrays.asList(new Melon("Gac", 2000),
    new Melon("Horned", 1600), new Melon("Apollo", 3000),
    new Melon("Gac", 3000), new Melon("hemi", 1600));
```

Мы намеренно добавили ошибку в последний объект `Melon`. На этот раз его поле `type` имеет тип `String`. Если мы применим компаратор `byWeightAndType`, то результат будет следующим:

```
Horned(1600g), hemi(1600g), ...
```

Будучи компаратором лексикографического порядка, `byWeightAndType` поставит `Horned` перед `hemi`. Поэтому будет полезно отсортировать дыни по сорту без учета регистра. Элегантное решение этой задачи будет опираться на еще одну разновидность метода `thenComparing()`, которая позволяет нам передавать функцию `Function` и компаратор `Comparator` в качестве аргументов. Переданная функция извлекает

ключ сортировки Comparable, и заданный компаратор используется для сравнения этого ключа сортировки:

```
Comparator<Melon> byWeightAndType =  
Comparator.comparing(Melon::getWeight)  
    .thenComparing(Melon::getType, String.CASE_INSENSITIVE_ORDER);
```

На этот раз результат будет следующим (мы снова на верном пути):

```
hemi(1600g), Horned(1600g), ...
```



Для значений типа `int`, `long` и `double` мы имеем методы `comparingInt()`, `comparingLong()`, `comparingDouble()`, `thenComparingInt()`, `thenComparingLong()` и `thenComparingDouble()`. Методы `comparing()` и `thenComparing()` поставляются в комплекте с теми же разновидностями.

Композиция функций

Лямбда-выражения, представляемые посредством интерфейса `Function`, могут выстраиваться в цепочку посредством методов `Function.andThen()` и `Function.compose()`.

Метод `andThen(Function<? super R,? extends V> after)` возвращает составную `Function`, которая делает следующее:

- ♦ применяет эту функцию к своему входу;
- ♦ применяет функцию `after` к результату.

Давайте посмотрим на ее пример:

```
Function<Double, Double> f = x -> x * 2;  
Function<Double, Double> g = x -> Math.pow(x, 2);  
Function<Double, Double> gf = f.andThen(g);  
double resultgf = gf.apply(4d); // 64.0
```

В этом примере функция `f` применяется к входу (4). Результат применения функции `f` равен 8 ($f(4) = 4 * 2$). И он является входом во вторую функцию, `g`. Результат применения функции `g` равен 64 ($g(8) = \text{Math.pow}(8, 2)$). Схема на рис. 9.19 демонстрирует процесс для четырех входов — 1, 2, 3 и 4.

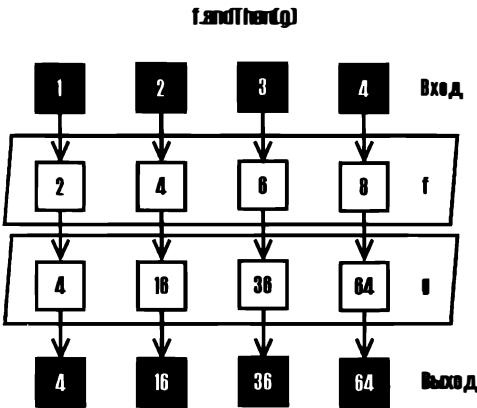


Рис. 9.19

Таким образом, это выглядит как $g(f(x))$. Противоположное, $f(g(x))$, может быть сформировано с помощью метода `Function.compose()`. Возвращенная составная функция применяет функцию `before` к своему входу, а затем применяет *эту функцию* к результату:

```
double resultfg = fg.apply(4d); // 32.0
```

В этом примере функция g применяется к входу (4). Результат применения функции g равен 16 ($g(4) = \text{Math.pow}(4, 2)$). Он и является входом во вторую функцию, f . Результат применения функции f равен 32 ($f(16) = 16 * 2$). Схема на рис. 9.20 демонстрирует процесс для четырех входов — 1, 2, 3 и 4.

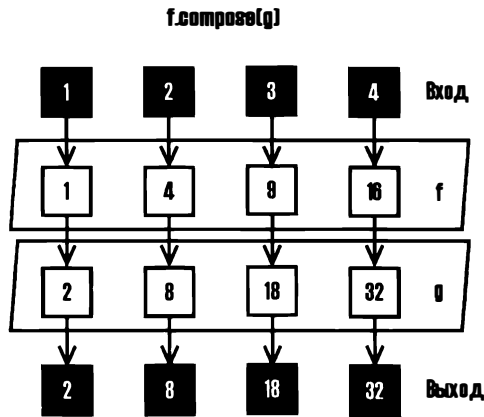


Рис. 9.20

Основываясь на тех же принципах, мы можем разработать приложение для редактирования статьи путем композиции методов `addIntroduction()`, `addBody()` и `addConclusion()`. Взгляните на исходный код, который прилагается к этой книге, чтобы увидеть его имплементацию.

Мы можем написать и другие конвейеры, просто жонглируя этим процессом композиции.

198. Методы по умолчанию

Методы по умолчанию были добавлены в Java 8. Их основная цель — обеспечивать поддержку интерфейсов, с тем чтобы они могли эволюционировать за пределами абстрактного контракта (содержать только абстрактные методы). Это средство очень полезно для разработчиков, которые пишут библиотеки и хотят улучшать API, сохраняя совместимость с предыдущими версиями своих проектов. Посредством методов по умолчанию интерфейс можно обогащать без нарушения существующих имплементаций.

Метод по умолчанию имплементируется непосредственно в интерфейсе и распознается по ключевому слову `default`.

Например, следующий ниже интерфейс определяет абстрактный метод `area()` и метод `perimeter()` по умолчанию :

```
public interface Polygon {  
    public double area();  
  
    default double perimeter(double...segments) {  
        return Arrays.stream(segments)  
            .sum();  
    }  
}
```

Поскольку периметр всех распространенных многоугольников (например, квадратов) является суммой их сторон, мы можем имплементировать метод здесь. А вот формула площади различается от многоугольника к многоугольнику, и поэтому имплементация по умолчанию не будет такой полезной.

Теперь давайте определим класс `Square`, который имплементирует интерфейс `Polygon`. Его цель — выразить площадь квадрата через периметр:

```
public class Square implements Polygon {  
    private final double edge;  
  
    public Square(double edge) {  
        this.edge = edge;  
    }  
  
    @Override  
    public double area() {  
        return Math.pow(perimeter(edge, edge, edge, edge) / 4, 2);  
    }  
}
```

Другие многоугольники (например, прямоугольники и треугольники) могут имплементировать интерфейс `Polygon` и выразить площадь на основе периметра, вычисляемого посредством имплементации по умолчанию.

Однако в некоторых случаях нам может потребоваться переопределить имплементацию метода по умолчанию. Например, класс `Square` может переопределить метод `perimeter()` следующим образом:

```
@Override  
public double perimeter(double...segments) {  
    return segments[0] * 4;  
}
```

Мы можем вызвать его так:

```
@Override  
public double area() {  
    return Math.pow(perimeter(edge) / 4, 2);  
}
```

Резюме

Дело сделано! В настоящей главе рассмотрены бесконечные потоки, `null`-безопасные потоки и методы по умолчанию. Полный список задач охватывал группирование, разбиение и коллекторы, включая коллектор `JDK 12 teeing()` и написание собственного коллектора. Кроме того, были рассмотрены такие интересные темы, как применение методов `takeWhile()`, `dropWhile()`, композиция функций, предикатов и компараторов, тестирование и отладка лямбда-выражений и другие.

Вы можете скачать приложения из этой главы, чтобы увидеть результаты и получить дополнительные детали, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

10

Конкурентность — пулы нитей исполнения, объекты *Callable* и синхронизаторы

Эта глава содержит 14 задач с привлечением конкурентности (псевдоодновременной работы) в среде Java. Мы начнем с нескольких основополагающих примеров, связанных с жизненными циклами нитей исполнения и применением замкового механизма на уровне объектов и классов. Затем мы продолжим обсуждение рядом задач, касающихся пулов нитей исполнения в Java, включая пул обкрадывающих нитей исполнения JDK 8. После этого мы займемся задачами, посвященными интерфейсам *Callable* и *Future*. Далее мы посвятим несколько задач синхронизаторам Java (в частности, барьеру, семафору и обменнику).

К концу этой главы вы должны будете знать главные аспекты конкурентности Java и быть готовыми продолжить работу с целым рядом продвинутых задач.

Задачи

Используйте следующие задачи для проверки вашего умения программировать конкурентность. Перед тем как обратиться к решениям и скачать примеры программ, настоятельно рекомендуется изучить каждую задачу.

199. **Состояния жизненного цикла нити исполнения.** Написать несколько программ, которые фиксируют каждое состояние жизненного цикла нити исполнения.
200. **Использование замка на уровне объекта против использования на уровне класса.** Написать несколько примеров, иллюстрирующих применение замка на уровне объекта и на уровне класса с помощью синхронизации нитей исполнения.

201. **Пулы нитей исполнения в среде Java.** Краткий обзор пулов нитей исполнения в Java.
202. **Пул нитей исполнения с одной нитью.** Написать программу, которая симулирует сборочную линию по проверке и упаковке лампочек с участием двух работников.
203. **Пул нитей исполнения с фиксированным числом нитей.** Написать программу, которая симулирует сборочную линию по проверке и упаковке лампочек с участием нескольких работников.
204. **Пулы кэшированных и запланированных нитей исполнения.** Написать программу, которая симулирует сборочную линию по проверке и упаковке лампочек с участием работников, число которых варьируется по мере необходимости (например, адаптировать число упаковщиков (увеличить или уменьшить) для обработки входящего наплыва работы, производимого контролером).
205. **Пул обкрадывающих нитей исполнения.** Написать программу, которая опирается на пул нитей исполнения, крадущих работу. Точнее, написать программу, которая симулирует сборочную линию по проверке и упаковке лампочек следующим образом: проверка происходит днем, а упаковка — ночью. Процесс проверки приводит к очереди из 15 млн лампочек каждый день.
206. **Операции callable и future.** Написать программу, которая симулирует сборочную линию по проверке и упаковке лампочек с помощью операций Callable и Future.
207. **Инициирование нескольких операций callable.** Написать программу, которая симулирует сборочную линию по проверке и упаковке лампочек следующим образом: проверка происходит днем, а упаковка — ночью. Процесс проверки приводит к очереди из 100 лампочек каждый день. Процесс упаковки должен упаковывать и возвращать все лампочки сразу. Другими словами, мы должны предъявлять все операции типа Callable и дожидаться их завершения.
208. **Стопоры.** Написать программу, которая использует CountdownLatch (стопор с обратным отсчетом) для симулирования процесса запуска сервера. Сервер считается запущенным после запуска его внутренних служб. Службы могут запускаться конкурентно и не зависят друг от друга.
209. **Барьеры.** Написать программу, которая использует CyclicBarrier (циклический барьер) для симулирования процесса запуска сервера. Сервер считается запущенным после запуска его внутренних служб. Службы могут быть подготовлены к запуску конкурентно (эта работа является времязатратной), но они работают взаимозависимо-поэтому, как только они готовы к запуску, они должны быть запущены все сразу.
210. **Обменники.** Написать программу, которая симулирует использование обменника, сборочной линии по проверке и упаковке лампочек с участием двух работников. Работник (контролер) проверяет лампочки и складывает их в

корзину. Когда корзина наполняется, работник отдает ее другому работнику (упаковщику), от которого тот получает пустую корзину. Процесс повторяется до тех пор, пока конвейер не остановится.

211. **Семафоры.** Написать программу, которая симулирует использование в парикмахерской одного семафора в день. В общих чертах, наша парикмахерская может обслуживать максимум трех человек одновременно (в ней всего три кресла). Когда клиент приходит в парикмахерскую, он пытается занять место. После того как парикмахер его обслужит, клиент освобождает кресло. Если человек приходит в парикмахерскую, когда все три кресла заняты, то он должен подождать некоторое время. Если это время пройдет, и ни одно кресло не будет освобождено, то клиент покидает парикмахерскую.
212. **Фазировщики.** Написать программу, которая использует класс `Phaser` для симулирования процесса запуска сервера в три фазы. Сервер считается запущенным после запуска пяти внутренних служб. В первой фазе нам нужно одновременно запустить три сервиса. Во второй фазе — одновременно запустить еще две службы (они могут быть запущены только в том случае, если первые три уже запущены). В третьей фазе сервер выполняет заключительную сверку и считается запущенным.

Решения

В следующих разделах описаны решения перечисленных выше задач. Помните, что обычно не существует одного единственно правильного варианта решения той или иной задачи. Кроме того, имейте в виду, что приведенные здесь объяснения включают только самые интересные и важные детали, необходимые для решения. Вы можете скачать примеры решений, чтобы увидеть дополнительные детали и поэкспериментировать с программами, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

199. Состояния жизненного цикла нити исполнения

Состояния нити исполнения Java выражаются посредством подкласса `Thread.State` класса перечислений `Enum`. Возможные состояния нити исполнения Java показаны на рис. 10.1.

Ниже перечислены разные состояния жизненного цикла:

- ◆ состояние `NEW`;
- ◆ состояние `Runnable`;
- ◆ состояние `BLOCKED`;
- ◆ состояние `WAITING`;
- ◆ состояние `TIMED_WAITING`;
- ◆ состояние `TERMINATED`.

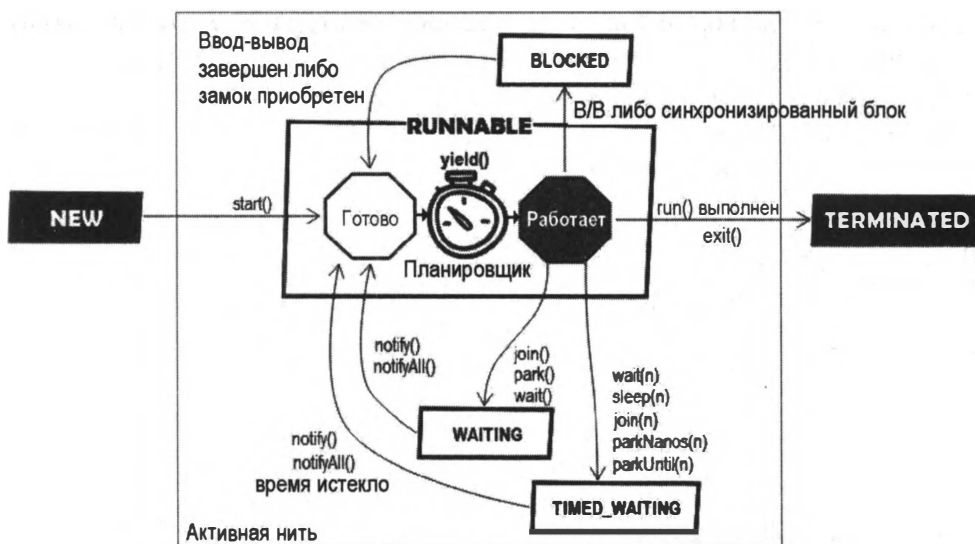


Рис. 10.1

Давайте разберемся во всех этих состояниях в следующих далее разделах.

Состояние **NEW**

Нить исполнения Java находится в состоянии **NEW**, если она создана, но не запущена (нитевой конструктор создает нити в состоянии **NEW**). Она находится в этом состоянии до тех пор, пока не будет вызван метод `start()`. Исходный код, прилагаемый к этой книге, содержит несколько фрагментов кода, которые раскрывают это состояние с помощью разных методов построения, включая лямбда-выражения. Для краткости приведем лишь одну из этих конструкций:

```
public class NewThread {
    public void newThread() {
        Thread t = new Thread(() -> {});
        System.out.println("NewThread: " + t.getState()); // NEW
    }
}
```

```
NewThread nt = new NewThread();
nt.newThread();
```

Состояние **RUNNABLE**

Переход из состояния **NEW** в состояние **RUNNABLE** осуществляется путем вызова метода `start()`. В этом состоянии нить исполнения может работать или быть готовой к работе. Когда нить готова к работе, она ждет выделение ей нитевым планировщиком JVM необходимых ресурсов и времени для работы. Как только процессор будет доступен, нитевой планировщик запустит нить.

Следующий ниже фрагмент кода должен напечатать `RUNNABLE`, т. к. мы печатаем состояние нити после вызова метода `start()`. Но из-за внутренних механизмов нитевого планировщика это не гарантируется:

```
public class RunnableThread {
    public void runnableThread() {
        Thread t = new Thread(() -> {});
        t.start();

        // RUNNABLE
        System.out.println("RunnableThread : " + t.getState());
    }
}

RunnableThread rt = new RunnableThread();
rt.runnableThread();
```

Состояние **BLOCKED**

Когда нить пытается исполнить операции ввода-вывода или синхронизированные блоки, она может войти в состояние `BLOCKED`. Например, если нить `t1` пытается войти в синхронизированный блок кода, к которому уже обращается другая нить `t2`, то `t1` удерживается в состоянии `BLOCKED` до тех пор, пока она не сможет приобрести замок.

Сценарий должен быть таким:

1. Создать две нити исполнения: `t1` и `t2`.
2. Запустить `t1` посредством метода `start()`:
 - `t1` исполнит метод `run()` и приобретет замок для синхронизированного метода `syncMethod()`;
 - метод `SyncMethod()` будет держать `t1` внутри вечно, т. к. он имеет бесконечный цикл.
3. Через две секунды (произвольное время) запустить `t2` посредством метода `start()`:
 - `t2` исполнит код метода `run()` и окажется в состоянии `BLOCKED`, т. к. не сможет приобрести замок метода `syncMethod()`.

Фрагмент кода выглядит следующим образом:

```
public class BlockedThread {
    public void blockedThread() {

        Thread t1 = new Thread(new SyncCode());
        Thread t2 = new Thread(new SyncCode());

        t1.start();
        Thread.sleep(2000);
```

```
t2.start();
Thread.sleep(2000);

System.out.println("BlockedThread t1: "
    + t1.getState() + "(" + t1.getName() + ")");
System.out.println("BlockedThread t2: "
    + t2.getState() + "(" + t2.getName() + ")");

System.exit(0);
}

private static class SyncCode implements Runnable {
    @Override
    public void run() {
        System.out.println("Нить " + Thread.currentThread().getName()
            + " находится в методе run()");
        syncMethod();
    }

    public static synchronized void syncMethod() {
        System.out.println("Нить " + Thread.currentThread().getName()
            + " находится в методе syncMethod()");

        while (true) {
            // t1 останется здесь навсегда, поэтому t2 блокируется
        }
    }
}

BlockedThread bt = new BlockedThread();
bt.blockedThread();
```

Вот возможный результат (имена нитей исполнения могут отличаться от приведенных здесь):

```
Нить Thread-0 находится в методе run()
Нить Thread-0 находится в методе syncMethod()
Нить Thread-1 находится в методе run()
BlockedThread t1: RUNNABLE(Thread-0)
BlockedThread t2: BLOCKED(Thread-1)
```

Состояние **WAITING**

Нить исполнения t1, которая ожидает (без периода таймаута) завершения другой нити t2, находится в состоянии WAITING.

Сценарий должен быть таким:

1. Создать нить исполнения `t1`.
2. Запустить `t1` посредством метода `start()`.
3. В методе `run()` из `t1`:
 - создать еще одну нить исполнения — `t2`;
 - запустить `t2` посредством метода `start()`;
 - пока `t2` работает, вызвать `t2.join()`; поскольку нити `t2` нужно присоединиться к нити `t1` (или, другими словами, `t1` должна ждать до тех пор, пока `t2` не умрет), `t1` находится в состоянии `WAITING`.
4. В методе `run()` нити `t2`, нить `t2` печатает состояние `t1`, которое должно быть `WAITING` (при печати состояния нити `t1`, нить `t2` работает, следовательно, нить `t1` ждет).

Фрагмент кода выглядит следующим образом:

```
public class WaitingThread {
    public void waitingThread() {
        new Thread(() -> {
            Thread t1 = Thread.currentThread();
            Thread t2 = new Thread(() -> {
                Thread.sleep(2000);
                System.out.println("WaitingThread t1: "
                    + t1.getState()); // WAITING
            });

            t2.start();

            t2.join();

        }).start();
    }
}
```

```
WaitingThread wt = new WaitingThread();
wt.waitingThread();
```

Состояние ***TIMED_WAITING***

Нить исполнения `t1`, которая ждет завершения другой нити исполнения `t2` в течение явно заданного таймаута, находится в состоянии `TIMED_WAITING`.

Сценарий должен быть таким:

1. Создать нить исполнения `t1`.
2. Запустить `t1` посредством метода `start()`.
3. В методе `run()` нити `t1` добавить время сна в две секунды (произвольное время).

Пока нить `t1` работает, главная нить печатает состояние нити `t1` — состояние должно быть `TIMED_WAITING`, т. к. `t1` находится в `sleep()` с таймаутом, который истекает через 2 секунды.

Фрагмент кода выглядит следующим образом:

```
public class TimedWaitingThread {
    public void timedWaitingThread() {
        Thread t = new Thread(() -> {
            Thread.sleep(2000);
        });

        t.start();

        Thread.sleep(500);

        System.out.println("TimedWaitingThread t: "
            + t.getState()); // TIMED_WAITING
    }
}

TimedWaitingThread twt = new TimedWaitingThread();
twt.timedWaitingThread();
```

Состояние **TERMINATED**

Нить исполнения, которая успешно завершает свою работу либо прерывается ненормальным образом, находится в состоянии `TERMINATED`. Это очень просто смоделировать, как в следующем ниже фрагменте кода (главная нить приложения печатает состояние нити `t`; когда это происходит, это означает, что нить `t` сделала свою работу):

```
public class TerminatedThread {
    public void terminatedThread() {
        Thread t = new Thread(() -> {});
        t.start();

        Thread.sleep(1000);

        System.out.println("TerminatedThread t: "
            + t.getState()); // TERMINATED
    }
}

TerminatedThread tt = new TerminatedThread();
tt.terminatedThread();
```

Для того чтобы писать нитебезопасные классы, мы можем рассмотреть следующие технические приемы:

- ◆ не иметь *состояния* (классы без экземплярных и статических переменных);

- ◆ иметь *состояние*, но не делиться им (например, использовать экземплярные переменные посредством объектов `Runnable`, `ThreadLocal` и т. д.);
- ◆ иметь *состояние*, но оно должно быть немутуируемым;
- ◆ использовать передачу сообщений (например, как каркас Akka);
- ◆ использовать синхронизированные блоки;
- ◆ использовать волатильные переменные;
- ◆ использовать структуры данных из пакета `java.util.concurrent`;
- ◆ использовать синхронизаторы (например, стопор `CountDownLatch` и барьер `Barrier`);
- ◆ использовать замки из пакета `java.util.concurrent.locks`.

200. Использование замка на уровне объекта против использования на уровне класса

В среде Java блок кода, помеченный как `synchronized`, может исполняться одной нитью в каждый момент времени. Поскольку Java является многонитевой средой (она поддерживает конкурентность, одновременность запуска), ей необходим механизм синхронизации во избежание проблем, специфичных для конкурентных сред (например, тупиков и нарушений согласованности памяти).

Нить исполнения может применять замки на уровне объекта или на уровне класса.

Замок на уровне объекта

Использовать замок на уровне объекта можно путем маркировки нестатического блока кода или нестатического метода (замкового объекта для объекта этого метода) ключевым словом `synchronized`. В следующих далее примерах только одной нити в каждый момент времени будет разрешено исполнять синхронизированный метод/блок на заданном экземпляре класса.

- ◆ Синхронизированный метод:

```
public class Class011 {  
    public synchronized void method011() {  
        ...  
    }  
}
```

- ◆ Синхронизированный блок кода:

```
public class Class011 {  
    public void method011() {  
        synchronized(this) {  
            ...  
        }  
    }  
}
```

◆ Еще один синхронизированный блок кода:

```
public class ClassOll {
    private final Object ollLock = new Object();
    public void methodOll() {
        synchronized(ollLock) {
            ...
        }
    }
}
```

Замок на уровне класса

Для того чтобы защитить статические данные, применить замок на уровне класса можно путем маркировки статического метода/блока либо посредством приобретения замка на ссылке `.class` с помощью `synchronized`. В следующих далее примерах только одной нити исполнения одного из доступных во время выполнения экземпляров будет разрешено исполнять синхронизированный блок в каждый момент времени.

◆ Синхронизированный статический метод:

```
public class ClassCll {
    public synchronized static void methodCll() {
        ...
    }
}
```

◆ Синхронизированный блок и замок на `.class`:

```
public class ClassCll {
    public void method() {
        synchronized(ClassCll.class) {
            ...
        }
    }
}
```

◆ Синхронизированный блок кода и замок на каком-либо другом статическом объекте:

```
public class ClassCll {
    private final static Object aLock = new Object();

    public void method() {
        synchronized(aLock) {
            ...
        }
    }
}
```

Важно знать

Вот некоторые распространенные случаи, их которых следуют синхронизации.

- ◆ Две нити могут одновременно исполнять синхронизированный статический метод и нестатический метод одного и того же класса (см. класс `O11AndC11` приложения `P200_ObjectVsClassLevelLocking`). Это работает, потому что нити исполнения приобретают замки на разных объектах.
- ◆ Две нити не могут одновременно исполнять два разных синхронизированных статических метода (или один и тот же синхронизированный статический метод) одного и того же класса (см. класс `TwoC11` приложение `P200_ObjectVsClassLevelLocking`). Это не работает, потому что первая нить приобретает замок уровня класса. На выходе будут даны комбинации `staticMethod1():Thread-0`, следовательно, только одна нить исполняет только один статический синхронизированный метод:

```
TwoC11 instance1 = new TwoC11();  
TwoC11 instance2 = new TwoC11();
```

- ◆ Две нити, два экземпляра:

```
new Thread() -> {  
    instance1.staticMethod1();  
}).start();
```

```
new Thread() -> {  
    instance2.staticMethod2();  
}).start();
```

- ◆ Две нити, один экземпляр:

```
new Thread() -> {  
    instance1.staticMethod1();  
}).start();
```

```
new Thread() -> {  
    instance1.staticMethod2();  
}).start();
```

- ◆ Две нити могут одновременно исполнять несинхронизированные, синхронизированные статические и синхронизированные нестатические методы (см. класс `O11C11AndNoLock` приложения `P200_ObjectVsClassLevelLocking`).
- ◆ Можно безопасно вызывать синхронизированный метод из другого синхронизированного метода того же класса, который требует такого же замка. Это работает, потому что механизм `synchronized` является повторно входимым (если это тот же самый замок, то замок, приобретенный для первого метода, используется и во втором методе). См. класс `TwoSyncs` из приложения `P200_ObjectVsClassLevelLocking`.



В качестве общего правила нужно запомнить, что ключевое слово `synchronized` можно использовать только со статическими/нестатическими методами (не конструкторами)/блоками кода. Следует избегать синхронизации нефинальных полей и строковых литералов (с экземплярами типа `String`, созданными с помощью `new`, все в порядке).

201. Пулы нитей исполнения в среде Java

Пул нитей исполнения — это коллекция нитей, которая может использоваться для исполнения операций. Пул нитей исполнения отвечает за управление созданием, выделением и жизненными циклами своих нитей и способствует повышению производительности. Теперь поговорим об исполнителях.

Интерфейс *Executor*

В пакете `java.util.concurrent` имеется связка интерфейсов, предназначенных для исполнения операций. Самый простой из них называется `Executor`. Этот интерфейс выставляет наружу один метод с именем `execute(Runnable command)`. Вот пример исполнения одной операции с использованием этого метода:

```
public class SimpleExecutor implements Executor {
    @Override
    public void execute(Runnable r) {
        (new Thread(r)).start();
    }
}
```

```
SimpleExecutor se = new SimpleExecutor();
```

```
se.execute(() -> {
    System.out.println(
        "Простая операция, исполняемая посредством интерфейса Executor");
});
```

Интерфейс *ExecutorService*

Более сложным и всеобъемлющим интерфейсом, который выставляет наружу целый ряд дополнительных методов, является `ExecutorService`. Он является расширенной версией интерфейса `Executor`. Java идет в комплекте с полноценной имплементацией интерфейса `ExecutorService`, называемой `ThreadPoolExecutor`. Это пул нитей исполнения, который создается с помощью связки аргументов, как показано ниже:

```
ThreadPoolExecutor(
    int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
```

```
TimeUnit unit,  
BlockingQueue<Runnable> workQueue,  
ThreadFactory threadFactory,  
RejectedExecutionHandler handler)
```

Вот краткое описание каждого аргумента, указанного в этом коде:

- ◆ `corePoolSize` — число нитей исполнения, хранимых в пуле, даже если они простаивают (если не задано `allowCoreThreadTimeOut`);
- ◆ `maximumPoolSize` — максимальное число разрешенных нитей исполнения;
- ◆ `keepAliveTime` — время, по истечении которого неработающие нити исполнения будут удалены из пула (это неработающие нити, число которых превышает `corePoolSize`);
- ◆ `unit` — единица времени для аргумента `keepAliveTime`;
- ◆ `workQueue` — очередь для хранения экземпляров `Runnable` (только операций `Runnable`, предъявленных методом `execute()`) перед их исполнением;
- ◆ `threadFactory` — фабрика, используемая, когда исполнитель создает новую нить исполнения;
- ◆ `handler` — обработчик, контролирующий и принимающий решение в ситуации, когда `ThreadPoolExecutor` не может исполнить операцию `Runnable` из-за насыщения, а это происходит, когда границы нитей и емкости очереди заполнены (например, рабочая очередь `workQueue` имеет фиксированный размер, и размер `maximumPoolSize` тоже установлен).

Для того чтобы оптимизировать размер пула, нам необходимо собрать следующую информацию:

- ◆ число процессоров (`Runtime.getRuntime().availableProcessors()`);
- ◆ целевую задействованность процессора (в диапазоне `[0; 1]`);
- ◆ время ожидания (`w`);
- ◆ вычислительное время (`c`).

Следующая формула помогает определить оптимальный размер пула:

Число нитей исполнения

$$= \text{Число процессоров} * \text{Целевая задействованность процессора} * (1 + W/C)$$



TIP

Запомните, что для операций с интенсивными вычислениями (обычно малых операций), неплохо провести сравнительный анализ пула нитей исполнения с числом нитей, равным числу процессоров или числу процессоров + 1 (во избежание потенциальных пауз). Для трудоемких и блокирующих операций (например, ввода-вывода) лучше использовать более крупный пул, т. к. нити исполнения не будут доступны для планирования с высоким темпом. Кроме того, обратите внимание на взаимодействие с другими пулами (например, пулами соединений с базой данных и пулами соединений с сокетом).

Давайте рассмотрим пример исполнителя ThreadPoolExecutor:

```
public class SimpleThreadPoolExecutor implements Runnable {
    private final int taskId;

    public SimpleThreadPoolExecutor(int taskId) {
        this.taskId = taskId;
    }

    @Override
    public void run() {
        Thread.sleep(2000);
        System.out.println("Исполнение операции " + taskId
            + " посредством " + Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        BlockingQueue<Runnable> queue = new LinkedBlockingQueue<>(5);
        final AtomicInteger counter = new AtomicInteger();

        ThreadFactory threadFactory = (Runnable r) -> {
            System.out.println("Создание новой нити Cool-Thread-"
                + counter.incrementAndGet());

            return new Thread(r, "Cool-Thread-" + counter.get());
        };

        RejectedExecutionHandler rejectedHandler
            = (Runnable r, ThreadPoolExecutor executor) -> {
            if (r instanceof SimpleThreadPoolExecutor) {
                SimpleThreadPoolExecutor task=(SimpleThreadPoolExecutor) r;
                System.out.println("Отклонение операции " + task.taskId);
            }
        };

        ThreadPoolExecutor executor = new ThreadPoolExecutor(10, 20, 1,
            TimeUnit.SECONDS, queue, threadFactory, rejectedHandler);

        for (int i = 0; i < 50; i++) {
            executor.execute(new SimpleThreadPoolExecutor(i));
        }

        executor.shutdown();
        executor.awaitTermination(
            Integer.MAX_VALUE, TimeUnit.MILLISECONDS);
    }
}
```

Метод `main()` запускает 50 экземпляров операции `Runnable`. Каждая операция `Runnable` спит в течение двух секунд и затем печатает сообщение. Рабочая очередь ограничена пятью экземплярами операции `Runnable`, число стержневых нитей исполнения ограничено 10, максимальное число нитей — числом 20, и таймаут — одной секундой. Возможный результат будет выглядеть следующим образом:

```
Создание новой нити Cool-Thread-1
...
Создание новой нити Cool-Thread-20
Отклонение операции 25
...
Отклонение операции 49
Исполнение операции 22 посредством Cool-Thread-18
...
Исполнение операции 12 посредством Cool-Thread-2
```

Интерфейс *ScheduledExecutorService*

Интерфейс `ScheduledExecutorService` — это интерфейс `ExecutorService`, который может планировать операции для исполнения после заданной задержки, либо исполнять периодически. Здесь у нас есть такие методы, как `schedule()`, `scheduleAtFixedRate()` и `scheduleWithFixedDelay()`. В то время как метод `schedule()` используется для однократных операций, методы `scheduleAtFixedRate()` и `scheduleWithFixedDelay()` применяются для периодических операций.

Пулы нитей исполнения посредством класса *Executors*

Еще один шаг, и мы вводим вспомогательный класс `Executors`. Этот класс выставляет наружу несколько типов пулов нитей исполнения, используя следующие методы:

- ◆ `newSingleThreadExecutor()` — пул нитей исполнения, управляющий только одной нитью с неограниченной очередью, которая исполняет только одну операцию за раз:
`ExecutorService executor = Executors.newSingleThreadExecutor();`
- ◆ `newCachedThreadPool()` — пул нитей исполнения, который создает новые нити и удаляет неработающие нити (через 60 секунд) по мере необходимости; размер стержневого пула равен 0, максимальный размер пула равен `Integer.MAX_VALUE` (этот пул нитей исполнения расширяется при увеличении спроса и сжимается при его уменьшении):
`ExecutorService executor = Executors.newCachedThreadPool();`
- ◆ `newFixedThreadPool()` — пул нитей исполнения с фиксированным числом нитей и неограниченной очередью, что создает эффект бесконечного таймаута (размер стержневого пула и максимальный размер пула равны указанному размеру):
`ExecutorService executor = Executors.newFixedThreadPool(5);`

- ◆ `newWorkStealingThreadPool()` — пул нитей исполнения, основанный на алгоритме кражи работы (он действует как слой над каркасом разветвления/соединения):

```
ExecutorService executor = Executors.newWorkStealingPool();
```

- ◆ `newScheduledThreadPool()` — пул нитей исполнения, который может планировать исполнение команд после заданной задержки либо исполнять их периодически (можно указать размер стержневого пула):

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(5);
```

202. Пул нитей исполнения с одной нитью

Для того чтобы показать, как работает пул нитей исполнения с одной нитью, допустим, что мы хотим написать программу, которая симулирует сборочную линию (или конвейер) для проверки и упаковки лампочек с участием двух работников.

Под *проверкой* мы понимаем, что работник выясняет работоспособность лампочки (горит/не горит). Под *упаковкой* мы понимаем действие, означающее, что работник берет проверенную лампочку и кладет ее в коробку. Этот вид процесса очень распространен практически на любой фабрике.

Вот эти два работника:

- ◆ так называемый производитель (или контролер — *checker*, *c*), который отвечает за проверку каждой лампочки, чтобы увидеть, что она горит;
- ◆ так называемый потребитель (или упаковщик — *packer*, *p*), который отвечает за упаковку каждой проверенной лампочки в коробку.

Такого рода задачи идеально подходят для шаблона архитектурного дизайна "производитель — потребитель", показанной на рис. 10.2.

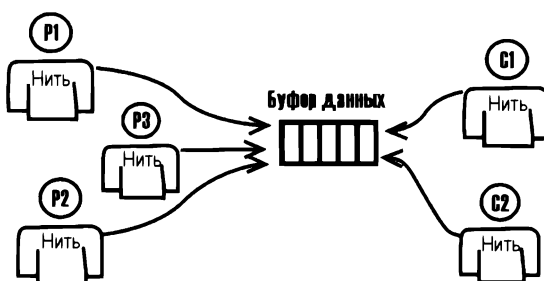


Рис. 10.2

Чаще всего в этом шаблоне производитель и потребитель взаимодействуют через очередь (производитель ставит данные в очередь, а потребитель изымает данные из очереди). Эта очередь называется *буфером данных*. Разумеется, в зависимости от конструкции указанного процесса роль буфера данных могут выполнять и другие структуры данных.

Теперь посмотрим, как имплементировать этот шаблон, если производитель ждет, пока потребитель не освободится.

Позже мы выполним имплементацию этого шаблона для производителя, который не ждет потребителя.

Производитель ждет, когда потребитель освободится

При запуске сборочной линии производитель будет проверять входящие лампочки одну за другой, в то время как потребитель будет упаковывать их (по одной лампочке в каждую коробку). Этот процесс повторяется до тех пор, пока конвейер не остановится.

Схема на рис. 10.3 является графическим представлением этого процесса между производителем и потребителем.

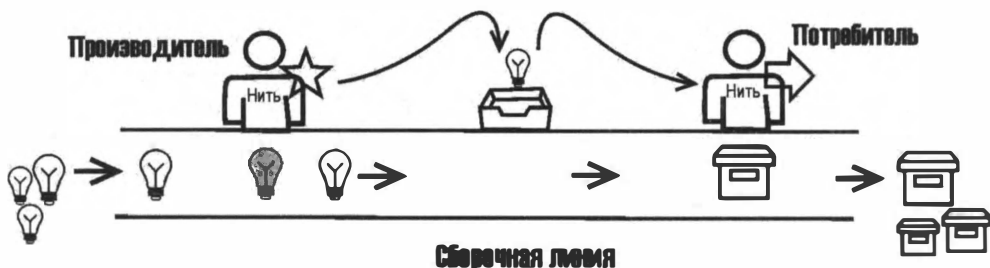


Рис. 10.3

Мы можем считать сборочную линию помощником нашей фабрики, поэтому ее можно имплементировать как вспомогательный или служебный класс (конечно, эту имплементацию можно легко переключить на нестатическую, поэтому не стесняйтесь переключиться, если в ваших случаях это имеет больше смысла):

```
public final class AssemblyLine {
    private AssemblyLine() {
        throw new AssertionError("There is a single assembly line!");
    }
    ...
}
```

Разумеется, этот сценарий можно имплементировать многими способами, но мы заинтересованы в использовании исполнительской службы `ExecutorService` среды Java, а точнее метода `Executors.newSingleThreadExecutor()`. Исполнитель, который использует одну рабочую нить, оперируя неограниченной очередью, создается как раз этим методом.

У нас есть только два работника, поэтому мы можем использовать два экземпляра исполнителя (один экземпляр `Executor` будет запрашивать производителя, а другой будет запрашивать потребителя). Таким образом, производитель будет одной нитью, и потребитель — другой нитью:

```
private static ExecutorService producerService;
private static ExecutorService consumerService;
```

Поскольку производитель и потребитель являются хорошими друзьями, они решают работать по простому сценарию:

- ◆ производитель проверяет лампочку и передает ее потребителю только в том случае, если потребитель не занят (если потребитель занят, производитель будет ждать некоторое время, пока потребитель не освободится);
- ◆ производитель не будет проверять следующую лампочку до тех пор, пока ему не удастся передать текущую лампочку потребителю;
- ◆ потребитель будет упаковывать каждую входящую лампочку как можно скорее.

Этот сценарий хорошо работает для очередей `TransferQueue` или `SynchronousQueue`, который выполняет этот процесс, очень похоже на вышеупомянутый сценарий. Давайте применим трансферную очередь `TransferQueue`. Она является имплементацией интерфейса `BlockingQueue`, в которой производители ждут до тех пор, пока потребители не получат элементы. Имплементации интерфейса `BlockingQueue` являются нитебезопасными:

```
private static final TransferQueue<String> queue
    = new LinkedTransferQueue<>();
```

Рабочий процесс между производителем и потребителем имеет дисциплину доступа "первым вошел, первым вышел" (FIFO: первая проверенная лампочка является первой упакованной лампочкой), поэтому связанная трансферная очередь `LinkedTransferQueue` может быть хорошим вариантом выбора.

После запуска сборочной линии производитель будет непрерывно проверять лампочки, поэтому мы можем имплементировать его как класс следующим образом:

```
private static final int MAX_PROD_TIME_MS = 5 * 1000;
private static final int MAX_CONS_TIME_MS = 7 * 1000;
private static final int TIMEOUT_MS = MAX_CONS_TIME_MS + 1000;
private static final Random rnd = new Random();
private static volatile boolean runningProducer;
...
private static class Producer implements Runnable {

    @Override
    public void run() {
        while (runningProducer) {
            try {
                String bulb = "bulb-" + rnd.nextInt(1000);
                Thread.sleep(rnd.nextInt(MAX_PROD_TIME_MS));

                boolean transfered = queue.tryTransfer(bulb,
                    TIMEOUT_MS, TimeUnit.MILLISECONDS);

                if (transfered) {
                    logger.info(() -> "Проверена: " + bulb);
                }
            }
        }
    }
}
```

```
    } catch (InterruptedException ex) {
        Thread.currentThread().interrupt();
        logger.severe(() -> "Исключение: " + ex);
        break;
    }
}
}
```

Таким образом, производитель передает проверенную лампочку потребителю посредством метода `tryTransfer()`. Если есть возможность передать элементы потребителю до истечения времени ожидания, то указанный метод это сделает.



Следует избегать использования метода `transfer()`, который может заблокировать нить исполнения навсегда.

Для того чтобы просимулировать время, затрачиваемое производителем на проверку лампочки, соответствующая нить исполнения будет спать случайное число секунд между 0 и 5 (5 секунд — это максимальное время, необходимое для проверки лампочки). Если потребитель после этого времени недоступен, то будет потрачено больше времени (в `tryTransfer()`) до тех пор, пока потребитель не освободится либо не истечет время ожидания.

С другой стороны, потребитель имплементируется с использованием еще одного класса, как показано ниже:

```
private static volatile boolean runningConsumer;
...
private static class Consumer implements Runnable {
    @Override
    public void run() {
        while (runningConsumer) {
            try {
                String bulb = queue.poll(MAX_PROD_TIME_MS, TimeUnit.MILLISECONDS);

                if (bulb != null) {
                    Thread.sleep(rnd.nextInt(MAX_CONS_TIME_MS));
                    logger.info(() -> "Упакована: " + bulb);
                }
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
                logger.severe(() -> "Исключение: " + ex);
                break;
            }
        }
    }
}
```

Потребитель может взять лампочку у производителя посредством метода `queue.take()`, который извлекает и удаляет голову этой очереди, ожидая при необходимости до тех пор, пока лампочка не станет доступной. Либо он может вызвать метод `poll()`, в котором извлекается и удаляется голова очереди, либо, если эта очередь пуста, он возвращает `null`. Но ни один из этих двух методов для нас не подходит. Если производитель недоступен, то потребитель может застрять в методе `take()`. Если же очередь пуста (производитель прямо сейчас проверяет текущую лампочку), то метод `poll()` будет вызываться снова и снова очень быстро, создавая фиктивное повторение. Решением этой проблемы является метод `poll(long timeout, TimeUnit unit)`. Он извлекает и удаляет голову этой очереди и ожидает в течение заданного времени таймаута, если потребуется, до тех пор, пока лампочка не станет доступной. Метод возвращает значение `null` только в том случае, если очередь является пустой по истечении времени ожидания.

Для того чтобы просимулировать время, которое потребитель тратит на упаковку лампочки, соответствующая нить исполнения будет спать случайное число секунд между 0 и 7 (7 секунд — это максимальное время, необходимое для упаковки лампочки).

Операция запуска производителя и потребителя является очень простой и выполняется в методе `startAssemblyLine()` следующим образом:

```
public static void startAssemblyLine() {
    if (runningProducer || runningConsumer) {
        logger.info("Сборочная линия уже работает...");
        return;
    }

    logger.info("\n\nЗапуск сборочной линии...");
    logger.info(() -> "Лампочки, оставшиеся после предыдущего прогона: \n"
        + queue + "\n\n");

    runningProducer = true;
    producerService = Executors.newSingleThreadExecutor();
    producerService.execute(producer);
    runningConsumer = true;
    consumerService = Executors.newSingleThreadExecutor();
    consumerService.execute(consumer);
}
```

Остановка сборочной линии — это деликатный процесс, который может быть решен с помощью разных сценариев. В общих чертах, когда сборочный конвейер останавливается, производитель должен проверить текущую лампочку как последнюю лампочку, а потребитель должен ее упаковать. Возможно, что производителю придется подождать, пока потребитель не упакует свою текущую лампочку, прежде чем он сможет передать последнюю лампочку; далее потребитель должен упаковать и эту лампочку.

Для того чтобы последовать этому сценарию, мы сначала останавливаем производителя, а затем потребителя:

```
public static void stopAssemblyLine() {
    logger.info("Остановка сборочной линии...");

    boolean isProducerDown = shutdownProducer();
    boolean isConsumerDown = shutdownConsumer();

    if (!isProducerDown || !isConsumerDown) {
        logger.severe("Произошло что-то ненормальное во время
            выключения сборочной линии!");

        System.exit(0);
    }

    logger.info("Сборочная линия была успешно остановлена!");
}

private static boolean shutdownProducer() {
    runningProducer = false;
    return shutdownExecutor(producerService);
}
```

```
private static boolean shutdownConsumer() {
    runningConsumer = false;
    return shutdownExecutor(consumerService);
}
```

Наконец, мы даем достаточно времени производителю и потребителю на то, чтобы остановиться нормально (без прерывания нитей исполнения). Это происходит в методе `shutdownExecutor()` следующим образом:

```
private static boolean shutdownExecutor(ExecutorService executor) {
    executor.shutdown();

    try {
        if (!executor.awaitTermination(TIMEOUT_MS * 2, TimeUnit.MILLISECONDS)) {
            executor.shutdownNow();
            return executor.awaitTermination(TIMEOUT_MS * 2, TimeUnit.MILLISECONDS);
        }

        return true;
    } catch (InterruptedException ex) {
        executor.shutdownNow();
        Thread.currentThread().interrupt();
        logger.severe(() -> "Исключение: " + ex);
    }

    return false;
}
```

Первое, что мы делаем, — это устанавливаем статическую переменную `runningProducer` равной `false`. В результате этого цикл `while(runningProducer)` будет прерван, и, следовательно, данная лампочка будет последней из проверенных. Далее мы иницилируем процедуру выключения для производителя.

В случае потребителя первое, что мы делаем, — это устанавливаем статическую переменную `runningConsumer` равной `false`. В результате этого цикл `while(runningConsumer)` будет прерван, и, следовательно, данная лампочка будет последней из упакованных. Далее мы иницилируем процедуру выключения для потребителя.

Рассмотрим возможное исполнение сборочной линии (выполняем код в течение 10 секунд):

```
AssemblyLine.startAssemblyLine();
Thread.sleep(10 * 1000);
AssemblyLine.stopAssemblyLine();
```

Возможный результат будет таким:

```
Запуск сборочной линии...
...
[2019-04-14 07:39:40] [INFO] Проверена: bulb-89
[2019-04-14 07:39:43] [INFO] Упакована: bulb-89
...
Остановка сборочной линии...
...
[2019-04-14 07:39:53] [INFO] Packed: bulb-322
Сборочная линия была успешно остановлена!
```



TIP

Вообще говоря, если на остановку сборочной линии (она действует так, как если бы была заблокирована) требуется много времени, то, вероятно, существует несбалансированный темп между числом производителей и потребителей и/или между временем производства и потребления. Вам, возможно, потребуется добавить или вычесть производителей или потребителей.

Производитель не ждет, когда потребитель освободится

Если производитель может проверять лампочки быстрее, чем потребитель способен их упаковывать, то, скорее всего, они решат использовать следующий рабочий процесс:

- ♦ производитель будет проверять лампочки одну за другой и заталкивать их в очередь;
- ♦ потребитель будет извлекать лампочки из очереди и упаковывать их.

Поскольку потребитель работает медленнее производителя, очередь будет содержать проверенные, но неупакованные лампочки (мы можем исходить из малой вероятности, что очередь будет пустой). На рис. 10.4 мы имеем производителя, по-

требителя и очередь, используемую для хранения проверенных, но неупакованных лампочек.

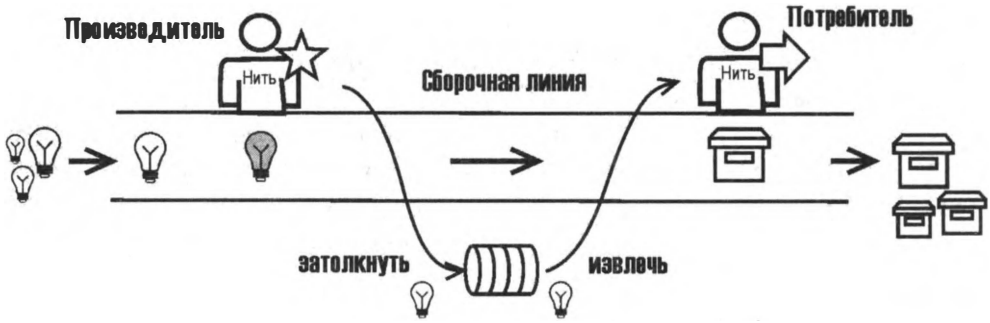


Рис. 10.4

В формировании этого сценария мы можем опереться на конкурентную связную очередь `ConcurrentLinkedQueue` (или связную блокирующую `LinkedBlockingQueue`). Это неограниченная нитебезопасная очередь, которая основана на связных узлах:

```
private static final Queue<String> queue = new ConcurrentLinkedQueue<>();
```

Для того чтобы втолкнуть лампочку в очередь, производитель вызывает метод `offer()`:

```
queue.offer(bulb);
```

С другой стороны, потребитель обрабатывает лампочки из очереди, используя метод `poll()` (поскольку потребитель медленнее производителя, случай, когда метод `poll()` возвращает `null`, будет редким):

```
String bulb = queue.poll();
```

Давайте запустим сборочную линию, дав ей поработать в течение первых 10 секунд. Это приведет к следующему результату:

Запуск сборочной линии...

...

[2019-04-14 07:44:58] [INFO] Проверена: bulb-827

[2019-04-14 07:44:59] [INFO] Проверена: bulb-257

[2019-04-14 07:44:59] [INFO] Упакована: bulb-827

...

Остановка сборочной линии...

...

[2019-04-14 07:45:08] [INFO] Проверена: bulb-369

[2019-04-14 07:45:09] [INFO] Упакована: bulb-690

...

Сборочная линия была успешно остановлена!

В этой точке сборочная линия остановлена, и в очереди мы имеем следующее (эти лампочки были проверены, но не упакованы):

```
[bulb-968, bulb-782, bulb-627, bulb-886, ...]
```

Мы перезапускаем сборочную линию и проверяем выделенные строки, которые показывают, что потребитель возобновляет свою работу с того места, где он остановился:

Запуск сборочной линии...

```
[2019-04-14 07:45:12] [INFO ] Упакована: bulb-968
[2019-04-14 07:45:12] [INFO ] Проверена: bulb-812
[2019-04-14 07:45:12] [INFO ] Проверена: bulb-470
[2019-04-14 07:45:14] [INFO ] Упакована: bulb-782
[2019-04-14 07:45:15] [INFO ] Проверена: bulb-601
[2019-04-14 07:45:16] [INFO ] Упакована: bulb-627
...
```

203. Пул нитей исполнения с фиксированным числом нитей

Эта задача повторяет сценарий из разд. 202 "Пул нитей исполнения с одной нитью". На этот раз сборочная линия использует трех производителей и двух потребителей (рис. 10.5).

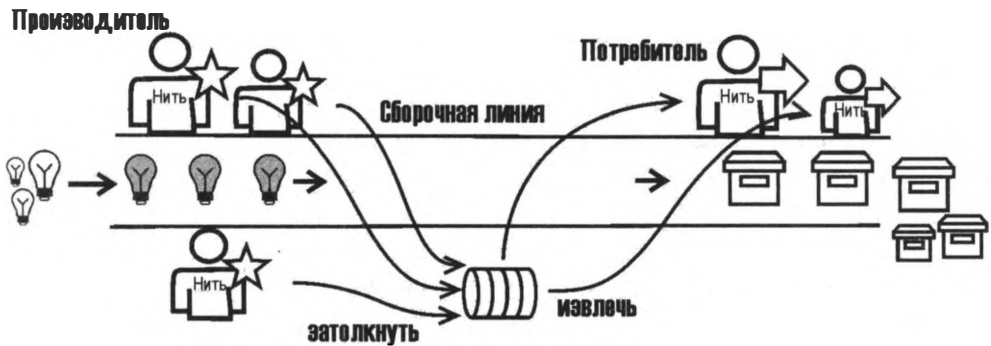


Рис. 10.5

Для симулирования фиксированного числа производителей и потребителей мы можем опереться на метод `Executors.newFixedThreadPool(int nThreads)`. Мы выделяем по одной нити на производителя (соответственно, потребителя), поэтому код довольно простой:

```
private static final int PRODUCERS = 3;
private static final int CONSUMERS = 2;
private static final Producer producer = new Producer();
private static final Consumer consumer = new Consumer();
private static ExecutorService producerService;
private static ExecutorService consumerService;
...
producerService = Executors.newFixedThreadPool (PRODUCERS);
for (int i = 0; i < PRODUCERS; i++) {
```



```

producerService.execute(producer);
}

consumerService = Executors.newFixedThreadPool(CONSUMERS);
for (int i = 0; i < CONSUMERS; i++) {
    consumerService.execute(consumer);
}

```

Очередь, в которую производители могут добавить проверенные лампочки, может иметь тип `LinkedTransferQueue` или `ConcurrentLinkedQueue` и т. д.

Полный исходный код, основанный на очередях `LinkedTransferQueue` и `ConcurrentLinkedQueue`, можно найти в архиве, прилагаемом к этой книге.

204. Пулы кэшированных и запланированных нитей исполнения

Эта задача повторяет сценарий из *разд. 202 "Пул нитей исполнения с одной нитью"*. На этот раз мы исходим из того, что производитель (может использоваться и несколько производителей) проверяет лампочку не более чем за одну секунду. Кроме того, потребителю (упаковщику) требуется максимум 10 секунд на то, чтобы упаковать лампочку. Время производителя и потребителя может быть сформировано следующим образом:

```

private static final int MAX_PROD_TIME_MS = 1 * 1000;
private static final int MAX_CONS_TIME_MS = 10 * 1000;

```

Очевидно, что в этих условиях один потребитель не может противостоять входящему наплыву работы. Очередь, используемая для хранения лампочек до тех пор, пока они не будут упакованы, будет непрерывно увеличиваться. Производитель будет добавлять лампочки в эту очередь гораздо быстрее, чем потребитель сможет изымать их из нее. Именно поэтому требуется больше потребителей, как показано на рис. 10.6.

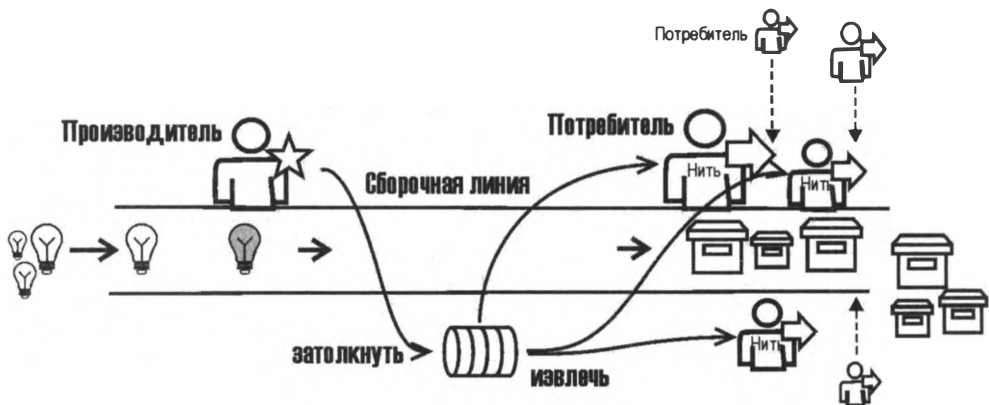


Рис. 10.6

Поскольку имеется единственный производитель, мы можем опереться на метод `Executors.newSingleThreadExecutor()`:

```
private static volatile boolean runningProducer;
private static ExecutorService producerService;
private static final Producer producer = new Producer();
...
public static void startAssemblyLine() {
    ...
    runningProducer = true;
    producerService = Executors.newSingleThreadExecutor();
    producerService.execute(producer);
    ...
}
```

Производитель `Producer` почти такой же, как и в предыдущих задачах, за исключением переменной `extraProdTime`:

```
private static int extraProdTime;
private static final Random rnd = new Random();
...
private static class Producer implements Runnable {
    @Override
    public void run() {
        while (runningProducer) {
            try {
                String bulb = "bulb-" + rnd.nextInt(1000);
                Thread.sleep(rnd.nextInt(MAX_PROD_TIME_MS) + extraProdTime);
                queue.offer(bulb);

                logger.info(() -> "Проверена: " + bulb);
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
                logger.severe(() -> "Исключение: " + ex);
                break;
            }
        }
    }
}
```

Переменная `extraProdTime` изначально равна 0. Она будет необходима, когда мы замедляем производство:

```
Thread.sleep(rnd.nextInt(MAX_PROD_TIME_MS) + extraProdTime);
```

Проработав некоторое время с высоким темпом, производитель устанет, и ему понадобится больше времени на то, чтобы проверять каждую лампочку. Если производитель замедляет темпы производства, то и число потребителей должно уменьшаться.

Когда производитель работает с высоким темпом, нам понадобится больше потребителей (упаковщиков). Но сколько их было? Использование фиксированного числа потребителей (`newFixedThreadPool()`) станет причиной как минимум двух недостатков:

- ◆ если производитель в какой-то момент замедлится, то некоторые потребители останутся без работы и будут просто торчать без дела;
- ◆ если производитель становится еще более эффективным, то требуется больше потребителей на то, чтобы противостоять входящему наплыву работы.

В сущности, мы должны иметь возможность варьировать число потребителей в зависимости от эффективности производителя.

Для таких заданий у нас есть метод `Executors.newCachedThreadPool()`. Пул кэшированных нитей исполнения будет использовать существующие нити повторно и создавать новые по мере необходимости (мы можем добавлять больше потребителей). Нити терминируются и удаляются из кэша, если они не были использованы в течение 60 секунд (мы можем удалять потребителей).

Начнем с единственного активного потребителя:

```
private static volatile boolean runningConsumer;
private static final AtomicInteger
    nrOfConsumers = new AtomicInteger();
private static final ThreadGroup threadGroup
    = new ThreadGroup("consumers");
private static final Consumer consumer = new Consumer();
private static ExecutorService consumerService;
...
public static void startAssemblyLine() {
    ...
    runningConsumer = true;
    consumerService = Executors
        .newCachedThreadPool((Runnable r) -> new Thread(threadGroup, r));
    nrOfConsumers.incrementAndGet();
    consumerService.execute(consumer);
    ...
}
```

Поскольку мы хотим иметь возможность видеть число нитей исполнения (потребителей) активных в некий момент, мы добавляем их в группу `ThreadGroup` посредством собственной фабрики `ThreadFactory`:

```
consumerService = Executors
    .newCachedThreadPool((Runnable r) -> new Thread(threadGroup, r));
```

Позже мы сможем получить число активных потребителей, используя следующий фрагмент кода:

```
threadGroup.activeCount();
```

Знание числа активных потребителей является хорошим индикатором, который может быть объединен с текущим размером очереди лампочек для определения потребности в большем числе потребителей.

Имплементация потребителя такова:

```
private static class Consumer implements Runnable {
    @Override
    public void run() {
        while (runningConsumer && queue.size() > 0
               || nrOfConsumers.get() == 1) {

            try {
                String bulb = queue.poll(MAX_PROD_TIME_MS
                                         + extraProdTime, TimeUnit.MILLISECONDS);

                if (bulb != null) {
                    Thread.sleep(rnd.nextInt(MAX_CONS_TIME_MS));
                    logger.info(() -> "Упакована: " + bulb + " потребителем: "
                                   + Thread.currentThread().getName());
                }
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
                logger.severe(() -> "Исключение: " + ex);
                break;
            }
        }

        nrOfConsumers.decrementAndGet();
        logger.warning(() -> "### Нить " + Thread.currentThread().getName()
                        + " возвращается в пул пока через 60 секунд!");
    }
}
```

Исходя из того что сборочная линия работает, потребитель будет продолжать упаковывать лампочки до тех пор, пока очередь не опустеет или он не останется единственным потребителем (у нас не может быть 0 потребителей). Мы можем трактовать пустую очередь, как означающую, что имеется слишком много потребителей. Таким образом, когда потребитель видит, что очередь пуста и он не единственный работающий потребитель, он становится бездействующим (через 60 секунд он будет автоматически удален из пула кэшированных нитей исполнения).

Не путайте переменную `nrOfConsumers` с методом `threadGroup.activeCount()`. Переменная `nrOfConsumers` хранит число потребителей (нитей), которые упаковывают лампочки прямо сейчас, в то время как метод `threadGroup.activeCount()` представляет всех активных потребителей (нити), включая те, которые не работают прямо сейчас (бездействуют) и просто ждут повторного использования или удаления из кэша.

Теперь возьмем реальный случай, когда инспектор будет наблюдать за сборочной линией, и когда он будет замечать, что текущее число потребителей не может противостоять входящему наплыву работы, он будет привлекать больше потребителей, чтобы те присоединялись к работе (допускается максимум 50 потребителей). Более того, когда он будет замечать, что некоторые потребители просто простаивают без дела, он будет отправлять их на другие рабочие места. Схема на рис. 10.7 является графическим представлением данного сценария.

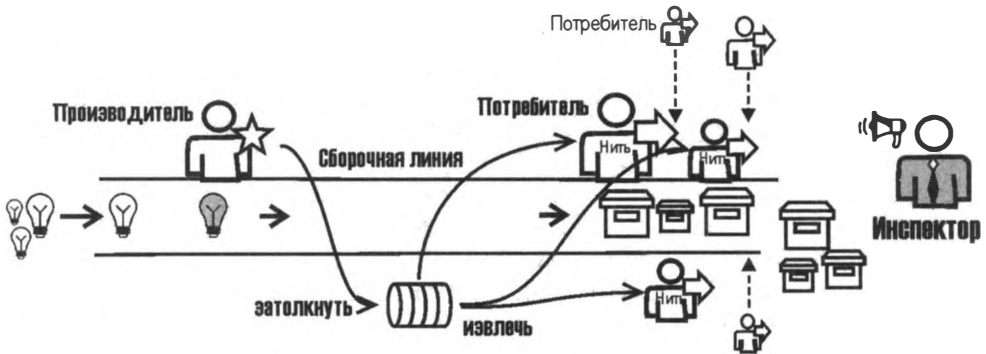


Рис. 10.7

Для целей тестирования наш инспектор, `newSingleThreadScheduledExecutor()`, будет однопоточным исполнителем, который может планировать выполнение заданных команд после указанной задержки. Он также может исполнять команды периодически:

```
private static final int MAX_NUMBER_OF_CONSUMERS = 50;
private static final int MAX_QUEUE_SIZE_ALLOWED = 5;
private static final int MONITOR_QUEUE_INITIAL_DELAY_MS = 5000;
private static final int MONITOR_QUEUE_RATE_MS = 3000;
private static ScheduledExecutorService monitorService;
...
private static void monitorQueueSize() {
    monitorService = Executors.newSingleThreadScheduledExecutor();

    monitorService.scheduleAtFixedRate(() -> {
        if (queue.size() > MAX_QUEUE_SIZE_ALLOWED
            && threadGroup.activeCount() < MAX_NUMBER_OF_CONSUMERS) {
            logger.warning("### Добавление нового потребителя (команды) ...");

            nrOfConsumers.incrementAndGet();
            consumerService.execute(consumer);
        }

        logger.warning(() -> "### Лампочки в очереди: " + queue.size()
            + " | Активные нити: " + threadGroup.activeCount()
            + " | Потребители: " + nrOfConsumers.get())
    }, MONITOR_QUEUE_INITIAL_DELAY_MS, MONITOR_QUEUE_RATE_MS);
}
```

```
+ " | Без дела: " + (threadGroup.activeCount()
    - nrOfConsumers.get()));
}, MONITOR_QUEUE_INITIAL_DELAY_MS, MONITOR_QUEUE_RATE_MS,
    TimeUnit.MILLISECONDS);
}
```

Мы опираемся на метод `scheduleAtFixedRate()` для мониторинга сборочной линии каждые три секунды с начальной задержкой в пять секунд. Таким образом, каждые три секунды инспектор проверяет размер очереди лампочек. Если в очереди находится более 5 лампочек и менее 50 потребителей, то инспектор просит нового потребителя подключиться к сборочной линии. Если очередь содержит 5 лампочек или менее либо уже 50 потребителей, то инспектор не предпринимает никаких действий.

Если мы сейчас запустим сборочную линию, то увидим, как число потребителей будет расти до тех пор, пока размер очереди не станет меньше шести. Возможный снимок будет выглядеть следующим образом:

Запуск сборочной линии...

```
[11:53:20] [INFO] Проверено: bulb-488
```

...

```
[11:53:24] [WARNING] ### Добавление нового потребителя (команды) ...
```

```
[11:53:24] [WARNING] ### Лампочки в очереди: 7
```

```
    | Активные нити: 2
```

```
    | Потребители: 2
```

```
    | Без дела: 0
```

```
[11:53:25] [INFO] Проверено: bulb-738
```

...

```
[11:53:36] [WARNING] ### Лампочки в очереди: 23
```

```
    | Активные нити: 6
```

```
    | Потребители: 6
```

```
    | Без дела: 0
```

...

Когда нитей исполнения становится больше, чем нужно, некоторые из них оказываются бездействующими. Если в течение 60 секунд они не получают работы, то удаляются из кэша. Если работа выполняется при отсутствии бездействующей нити, то будет создана новая нить. Этот процесс повторяется постоянно до тех пор, пока мы не заметим равновесия на сборочной линии. Через некоторое время все начнет успокаиваться и нужное число потребителей окажется в малом диапазоне (небольшие колебания). Это происходит потому, что производитель выдает данные со случайным темпом, ограниченным не более одной секундой.

Через некоторое время (например, через 20 секунд) давайте замедлим производителя на 4 секунды (в результате чего теперь лампочку можно проверить максимум за 5 секунд):

```
private static final int SLOW_DOWN_PRODUCER_MS = 20 * 1000;
private static final int EXTRA_TIME_MS = 4 * 1000;
```

Это можно сделать с помощью еще одного метода

`newSingleThreadScheduledExecutor()`, как показано ниже:

```
private static void slowdownProducer() {
    slowdownerService = Executors.newSingleThreadScheduledExecutor();

    slowdownerService.schedule(() -> {
        logger.warning("### Замедлить производителя...");
        extraProdTime = EXTRA_TIME_MS;
    }, SLOW_DOWN_PRODUCER_MS, TimeUnit.MILLISECONDS);
}
```

Это произойдет только один раз, через 20 секунд после запуска сборочной линии. Поскольку темп работы производителя был уменьшен на 4 секунды, нет необходимости иметь такое же число потребителей, чтобы поддерживать очередь максимум из пяти лампочек.

Данная ситуация прослеживается в приведенных ниже результатах (обратите внимание, что в некоторые моменты для обработки очереди требуется только один потребитель):

```
...
[11:53:36] [WARNING] ### Лампочки в очереди: 23
                | Активные нити: 6
                | Потребители: 6
                | Без дела: 0
...
[11:53:39] [WARNING] ### Замедлить производителя...
...
[11:53:56] [WARNING] ### Нить Thread-5 возвращается
                в пул пока через 60 секунд!
[11:53:56] [INFO] Упакована: bulb-346 потребителем: Thread-2
...
[11:54:36] [WARNING] ### Лампочки в очереди: 1
                | Активные нити: 12
                | Потребители: 1
                | Без дела: 11
...
[11:55:48] [WARNING] ### Лампочки в очереди: 3
                | Активные нити: 1
                | Потребители: 1
                | Без дела: 0
```

Сборочная линия была успешно остановлена!

Запуск инспектора происходит после запуска сборочной линии:

```
public static void startAssemblyLine() {
    ...
    monitorQueueSize();
    slowdownProducer();
}
```

Полное приложение доступно в исходном коде, прилагаемом к этой книге.



При использовании пулов кэшированных нитей исполнения обратите внимание на число нитей, создаваемых для размещения в памяти числа предъявленных операций. В то время как для пулов с одной нитью исполнения и пулов с фиксированным числом нитей исполнения мы управляем числом создаваемых нитей сами, пул кэшированных нитей может решить создать слишком много нитей. Неконтролируемое создание нитей может привести к быстрому исчерпанию ресурсов. По этой причине в системах, которые уязвимы, перед перегрузкой лучше опираться на пулы с фиксированным числом нитей исполнения.

205. Пул обкрадывающих нитей исполнения

Давайте сосредоточимся на процессе упаковки, который должен быть имплементирован посредством пула обкрадывающих нитей исполнения. Для начала обсудим, что такое пул обкрадывающих нитей исполнения, и сделаем это путем сравнения с классическим пулом нитей исполнения. Схема на рис. 10.8 демонстрирует принцип работы классического пула нитей исполнения.

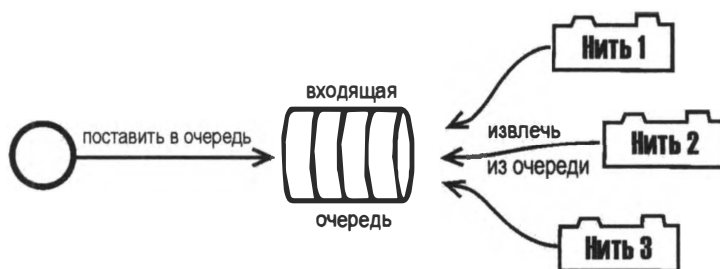


Рис. 10.8

Таким образом, пул нитей исполнения использует внутреннюю входящую очередь для хранения операций. Каждая нить должна извлекать операцию из очереди и исполнять ее. Это подходит для тех случаев, когда операции отнимают много времени и их число относительно невелико. Если же эти операции многочисленны и малы (их исполнение требует малое время), то также будет много конфликтов. Это не очень хорошо, и даже если эта очередь является беззамковой, то эта проблема решается не полностью.

Для того чтобы уменьшить число конфликтов и повысить производительность, пул нитей исполнения может опираться на алгоритм кражи работы и на одну очередь в расчете на нить. В этом случае для всех операций имеются центральная входящая очередь и дополнительная очередь (так называемая локальная очередь операций) для каждой нити (рабочей нити), как показано на рис. 10.9.

Таким образом, каждая нить исполнения будет извлекать операции из центральной очереди и помещать их в свою очередь. Каждая нить исполнения имеет собственную локальную очередь операций. Когда нить хочет обработать операцию, она просто извлекает операцию из своей локальной очереди. До тех пор, пока ее ло-

кальная очередь не пуста, нить будет продолжать обрабатывать операции из нее, не беспокоя другие нити (никаких конфликтов с другими нитями). Когда ее локальная очередь окажется пустой (как в случае нити 2 на приведенной схеме), она попытается украсть (с помощью алгоритма кражи работы) операции из локальных очередей, принадлежащих другим нитям (например, нить 2 крадет операции из нити 3). Если она не находит ничего, что можно украсть, она обращается к совместной центральной входящей очереди.

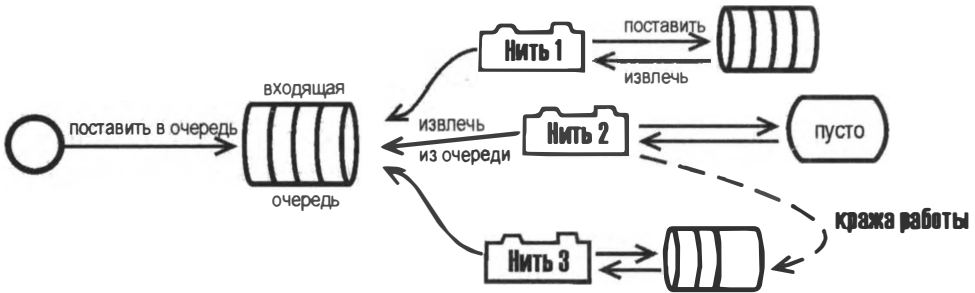


Рис. 10.9

Каждая локальная очередь фактически является двухсторонней очередью (queue, от англ. *double-ended queue*), поэтому к ней можно эффективно обращаться с обоих концов. Нить исполнения видит свою двустороннюю очередь как стек, и это означает, что она будет ставить в очередь (добавлять новые операции) и изымать из очереди (принимать операции для обработки) только с одного конца. Когда нить попытается украсть операции из очереди другой нити, она обратится к другому концу (например, нить 2 крадет из очереди нити 3 с другого конца). Таким образом, операции обрабатываются с одного конца и крадутся с другого.

Если две нити исполнения пытаются украсть из одной и той же локальной очереди, то возникает конфликт, но в нормальных условиях эта ситуация должна быть незначительной.

То, что мы только что описали, представляет собой каркас разветвления/соединения (fork/join), введенный в JDK 7 и приведенный в качестве примера в разд. "214. Каркас разветвления/соединения" главы 11. Начиная с JDK 8, класс `Executors` дополнен пулом обкрадывающих нитей исполнения, использующих число доступных процессоров в качестве целевого уровня параллелизма. Он доступен посредством методов `Executors.newWorkStealingPool()` и `Executors.newWorkStealingPool(int parallelism)`.

Давайте взглянем на исходный код этого пула нитей исполнения:

```
public static ExecutorService newWorkStealingPool() {
    return new ForkJoinPool(Runtime.getRuntime().availableProcessors(),
        ForkJoinPool.defaultForkJoinWorkerThreadFactory,
        null, true);
}
```

Таким образом, внутренне этот пул нитей исполнения создает экземпляр класса ForkJoinPool посредством следующего конструктора:

```
public ForkJoinPool(int parallelism,  
    ForkJoinPool.ForkJoinWorkerThreadFactory factory,  
    Thread.UncaughtExceptionHandler handler,  
    boolean asyncMode)
```

Мы имеем уровень параллелизма, установленный в `availableProcessors()`, нитевую фабрику по умолчанию и возвращающую новые нити, обработчик исключений `Thread.UncaughtExceptionHandler`, переданный как `null`, и режим `asyncMode`, установленный в `true`. Установка режима `asyncMode` равным `true` означает, что он мобилизует локальный режим планирования с дисциплиной доступа "первым вошел, первым вышел" (FIFO) для операций, которые разветвляются и никогда не соединяются. Этот режим может быть более подходящим, чем режим по умолчанию (основанный на локальном стеке) в программах, которые опираются на рабочие нити исполнения для обработки только событийных асинхронных операций.

Тем не менее не забывайте, что локальная очередь операций и алгоритм кражи работы мобилизуются только в том случае, если рабочие нити планируют новые операции в собственных локальных очередях. В противном случае пул ForkJoinPool — это просто исполнитель `ThreadPoolExecutor` с дополнительными накладными расходами.

При работе непосредственно с пулом ForkJoinPool мы можем поручать операциям явно планировать новые операции во время исполнения, используя объект `ForkJoinTask` (как правило, посредством `RecursiveTask` или `RecursiveAction`).

Но поскольку метод `newWorkStealingPool()` является более высоким уровнем абстракции для пула ForkJoinPool, мы не можем поручать операциям явно планировать новые операции во время исполнения. Поэтому метод `newWorkStealingPool()` будет решать внутренне, как работать, основываясь на операциях, которые мы проходим. Мы можем попробовать провести сравнение между методами `newWorkStealingPool()`, `newCachedThreadPool()` и `newFixedThreadPool()`, а также посмотреть, как они работают в двух сценариях:

- ◆ для большого числа мелких операций;
- ◆ для малого числа времязатратных операций.

Давайте рассмотрим решения для обоих сценариев в следующих далее разделах.

Большое число мелких операций

Поскольку производители (контролеры) и потребители (упаковщики) не работают одновременно, мы можем легко заполнить очередь под завязку 15 млн лампочек посредством тривиального цикла `for` (данная часть сборочной линии нас не очень интересует). Это показано в следующем ниже фрагменте кода:

```
private static final Random rnd = new Random();  
private static final int MAX_PROD_BULBS = 15_000_000;  
private static final BlockingQueue<String> queue
```

```
= new LinkedBlockingQueue<>();
...
private static void simulatingProducers() {
    logger.info("Симулирование работы производителей ночью...");
    logger.info(() -> "Производители проверили "
        + MAX_PROD_BULBS + " лампочек...");

    for (int i = 0; i < MAX_PROD_BULBS; i++) {
        queue.offer("bulb-" + rnd.nextInt(1000));
    }
}
```

Далее давайте создадим пул по умолчанию обкрадывающих нитей исполнения:

```
private static ExecutorService consumerService
    = Executors.newWorkStealingPool();
```

Для сравнения мы также будем использовать следующие ниже пулы нитей исполнения:

◆ пул кэшированных нитей исполнения:

```
private static ExecutorService consumerService
    = Executors.newCachedThreadPool();
```

◆ пул с фиксированным числом нитей исполнения, использующий число доступных процессоров в качестве числа нитей (число процессоров используется пулом по умолчанию обкрадывающих нитей исполнения в качестве уровня параллелизма):

```
private static final Consumer consumer = new Consumer();
private static final int PROCESSORS
    = Runtime.getRuntime().availableProcessors();
private static ExecutorService consumerService
    = Executors.newFixedThreadPool(PROCESSORS);
```

И начнем с 15 млн малых операций:

```
for (int i = 0; i < queueSize; i++) {
    consumerService.execute(consumer);
}
```

Класс `Consumer` обертывает простую операцию `queue.poll()`, поэтому она должна выполняться довольно быстро, как показано в следующем ниже фрагменте кода:

```
private static class Consumer implements Runnable {
    @Override
    public void run() {
        String bulb = queue.poll();

        if (bulb != null) {
            // ничего
        }
    }
}
```

График на рис. 10.10 представляет данные, собранные за 10 прогонов.

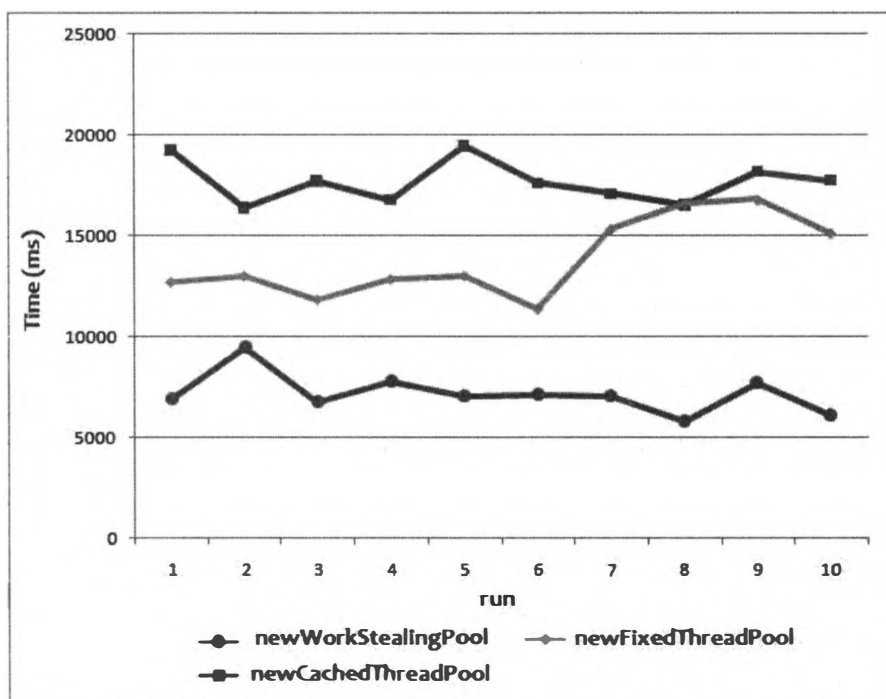


Рис. 10.10

Даже если этот сравнительный анализ не является профессиональным, мы можем видеть, что пул обкрадывающих нитей исполнения получил лучшие результаты (—●—), тогда как пул кэшированных нитей исполнения имеет худшие результаты (—■—).

Малое число времязатратных операций

Вместо того чтобы заполнять очередь 15 млн лампочек, давайте заполним 15 очередей по 1 тыс. лампочек в каждой:

```
private static final int MAX_PROD_BULBS = 15_000_000;
private static final int CHUNK_BULBS = 1_000_000;
private static final Random rnd = new Random();
private static final Queue<BlockingQueue<String>> chunks
    = new LinkedBlockingQueue<>();
...
private static Queue<BlockingQueue<String>> simulatingProducers() {
    logger.info("Симулирование работы производителей ночью...");
    logger.info(() -> "Производитель проверил "
        + MAX_PROD_BULBS + " лампочек...");
```

```
int counter = 0;
while (counter < MAX_PROD_BULBS) {
    BlockingQueue chunk = new LinkedBlockingQueue<>(CHUNK_BULBS);

    for (int i = 0; i < CHUNK_BULBS; i++) {
        chunk.offer("bulb-" + rnd.nextInt(1000));
    }

    chunks.offer(chunk);
    counter += CHUNK_BULBS;
}

return chunks;
}
```

И запустим 15 операций, используя следующий ниже код:

```
while (!chunks.isEmpty()) {
    Consumer consumer = new Consumer(chunks.poll());
    consumerService.execute(consumer);
}
```

Каждый потребитель перебирает 1 млн лампочек в цикле, используя вот этот исходный код:

```
private static class Consumer implements Runnable {

    private final BlockingQueue<String> bulbs;

    public Consumer(BlockingQueue<String> bulbs) {
        this.bulbs = bulbs;
    }

    @Override
    public void run() {
        while (!bulbs.isEmpty()) {
            String bulb = bulbs.poll();

            if (bulb != null) {}
        }
    }
}
```

График на рис. 10.11 представляет данные, собранные за 10 прогонов.

На этот раз, похоже, что пул обкрадывающих нитей исполнения (—●—) работал как пул обычных нитей исполнения.

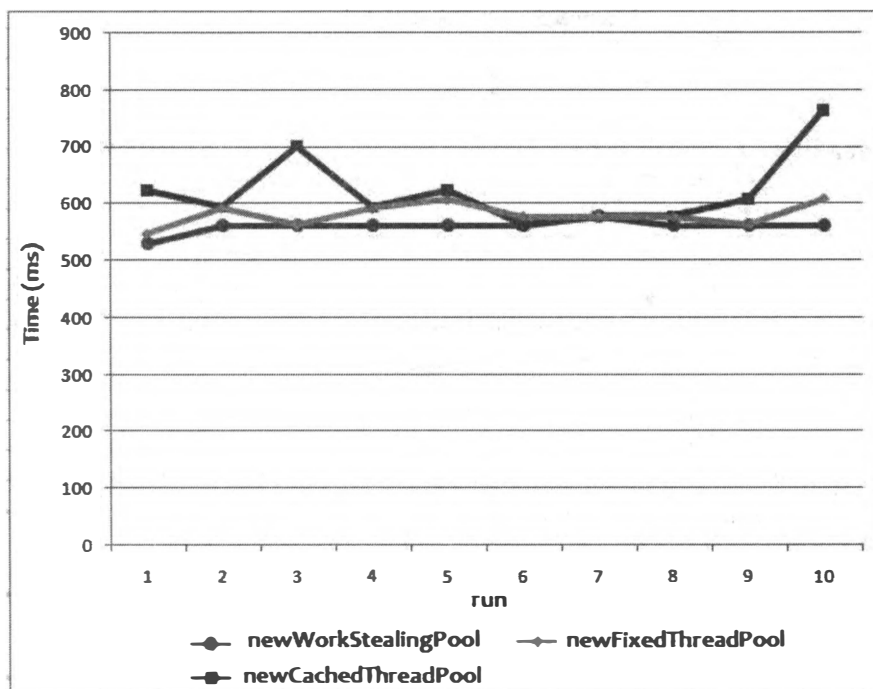


Рис. 10.11

206. Операции *Callable* и *Future*

Эта задача повторяет сценарий из разд. 202 "Пул нитей исполнения с одной нитью" ранее в этой главе. Мы хотим иметь одного производителя и одного потребителя, которые следуют такому сценарию:

1. Автоматическая система посылает запрос производителю, говоря: "Проверь эту лампочку, и если она в порядке, то верни ее мне; в противном случае скажи мне, что с этой лампочкой не так".
2. Автоматическая система ждет, когда производитель проверит лампочку.
3. Когда автоматическая система получает проверенную лампочку, та затем передает лампочку дальше потребителю (упаковщику) и повторяет процесс.
4. Если лампочка имеет дефект, то производитель выбрасывает исключение (`DefectBulbException`), и автоматическая система инспектирует причину проблемы.

Этот сценарий показан на рис. 10.12.

Для того чтобы сформировать этот сценарий, производитель должен иметь возможность возвращать результат и выбрасывать исключение. Поскольку наш производитель является `Runnable`, он не может делать ни того, ни другого. Но Java определяет интерфейс, который называется `Callable`. Это функциональный интерфейс с методом `call()`. В отличие от метода `run()` интерфейса `Runnable`, метод `call()` может возвращать результат и даже выбрасывать исключение — `V call() throws Exception`.

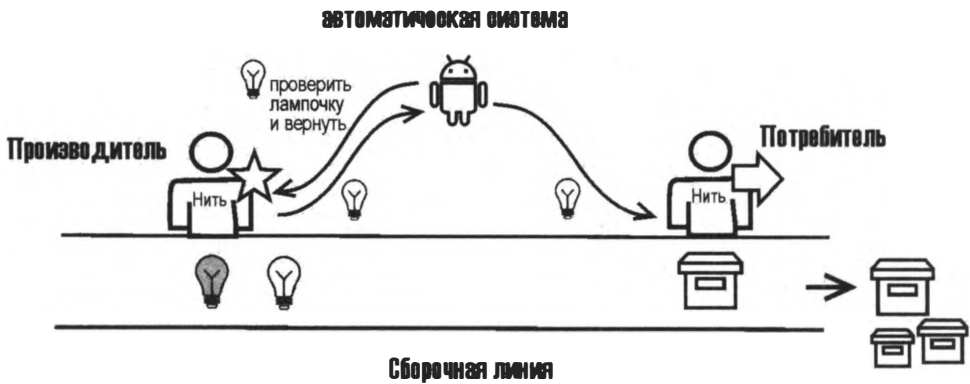


Рис. 10.12

Это означает, что производителя (контролера) можно написать следующим образом:

```
private static volatile boolean runningProducer;
private static final int MAX_PROD_TIME_MS = 5 * 1000;
private static final Random rnd = new Random();
...
private static class Producer implements Callable {
    private final String bulb;

    private Producer(String bulb) {
        this.bulb = bulb;
    }

    @Override
    public String call()
        throws DefectBulbException, InterruptedException {

        if (runningProducer) {
            Thread.sleep(rnd.nextInt(MAX_PROD_TIME_MS));

            if (rnd.nextInt(100) < 5) {
                throw new DefectBulbException("Дефект: " + bulb);
            } else {
                logger.info(() -> "Проверена: " + bulb);
            }

            return bulb;
        }

        return "";
    }
}
```

Исполнительная служба может предъявить операцию объекту `Callable` посредством метода `submit()`, но она не знает, когда будет доступен результат предъявленной операции.

Поэтому объект `Callable` немедленно возвращает особый тип под названием `Future`. Результат асинхронного вычисления представляется объектом типа `Future` — через объект типа `Future` мы можем получать результат операции, когда он становится доступным. В концептуальном плане, мы можем думать о типе `Future` как об обещании JavaScript или как о результате вычисления, которое будет выполнено в более поздний момент времени.

Теперь создадим производителя и предъявим его объекту типа `Callable`:

```
String bulb = "bulb-" + rnd.nextInt(1000);  
Producer producer = new Producer(bulb);
```

```
Future<String> bulbFuture = producerService.submit(producer);  
// эта строка кода выполняется немедленно
```

Поскольку объект типа `Callable` немедленно возвращает экземпляр типа `Future`, мы можем выполнять другие операции, ожидая результата предъявленной операции (флаговый метод `isDone()` возвращает `true`, если эта операция завершена):

```
while (!future.isDone()) {  
    System.out.println("Сделать что-то еще...");  
}
```

Извлечь результат из объекта типа `Future` можно с помощью блокирующего метода `Future.get()`. Этот метод блокирует до тех пор, пока не появится результат или не истечет указанный таймаут (если результат не будет доступен до таймаута, то будет выброшено исключение `TimeoutException`):

```
String checkedBulb = bulbFuture.get(  
    MAX_PROD_TIME_MS + 1000, TimeUnit.MILLISECONDS);
```

```
// эта строка выполняется только после того, как появится результат
```

Как только результат будет доступен, мы можем передать его потребителю `Consumer`, а также еще одну операцию производителю `Producer`. Этот цикл повторяется все время, пока потребитель и производитель работают. Соответствующий исходный код выглядит следующим образом:

```
private static void automaticSystem() {  
    while (runningProducer && runningConsumer) {  
        String bulb = "bulb-" + rnd.nextInt(1000);  
  
        Producer producer = new Producer(bulb);  
        Future<String> bulbFuture = producerService.submit(producer);  
        ...  
        String checkedBulb = bulbFuture.get(  
            MAX_PROD_TIME_MS + 1000, TimeUnit.MILLISECONDS);
```



```
    Consumer consumer = new Consumer(checkedBulb);
    if (runningConsumer) {
        consumerService.execute(consumer);
    }
}
...
}
```

Потребитель по-прежнему является Runnable, поэтому он не может возвращать результат или выбрасывать исключение:

```
private static final int MAX_CONS_TIME_MS = 3 * 1000;
...
private static class Consumer implements Runnable {
    private final String bulb;

    private Consumer(String bulb) {
        this.bulb = bulb;
    }

    @Override
    public void run() {
        if (runningConsumer) {
            try {
                Thread.sleep(rnd.nextInt(MAX_CONS_TIME_MS));
                logger.info(() -> "Упакована: " + bulb);
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
                logger.severe(() -> "Исключение: " + ex);
            }
        }
    }
}
}
```

Наконец, нам нужно запустить автоматическую систему. Соответствующий исходный код выглядит так:

```
public static void startAssemblyLine() {
    ...
    runningProducer = true;
    consumerService = Executors.newSingleThreadExecutor();

    runningConsumer = true;
    producerService = Executors.newSingleThreadExecutor();

    new Thread(() -> {
        automaticSystem();
    }).start();
}
```

Обратите внимание, что мы не хотим блокировать главную нить исполнения, поэтому мы запускаем автоматическую систему в новой нити. Благодаря этому главная нить исполнения может управлять процессом запуска-остановки сборочной линии.

Давайте запустим сборочную линию на несколько минут, чтобы собрать некоторую продукцию:

Запуск сборочной линии...

```
[08:38:41] [INFO ] Проверена: bulb-879
```

```
...
```

```
[08:38:52] [SEVERE ] Исключение: DefectBulbException: Defect: bulb-553
```

```
[08:38:53] [INFO ] Упакована: bulb-305
```

```
...
```

ОК, дело сделано! Давайте перейдем к последней теме.

Отмена операции *Future*

Операция `Future` может отменяться. Это достигается с помощью метода `cancel(Boolean mayInterruptIfRunning)`. Если мы передадим ему `true`, то нить, которая исполняет операцию, будет прервана; в противном случае нить может завершить операцию. Этот метод возвращает `true`, если операция была успешно отменена, в противном случае он возвращает `false` (в типичной ситуации по причине, что она уже завершилась нормально). Вот простой пример, который отменяет операцию, если на ее исполнение требуется более одной секунды:

```
long startTime = System.currentTimeMillis();
```

```
Future<String> future = executorService.submit(() -> {  
    Thread.sleep(3000);
```

```
    return "Операция завершена";  
});
```

```
while (!future.isDone()) {  
    System.out.println("Операция выполняется...");  
    Thread.sleep(100);
```

```
    long elapsedTime = (System.currentTimeMillis() - startTime);
```

```
    if (elapsedTime > 1000) {  
        future.cancel(true);  
    }  
}
```

Метод `isCancelled()` возвращает `true`, если операция была отменена до ее нормального завершения:

```
System.out.println("Операция была отменена: " + future.isCancelled()  
    + "\nОперация выполнена: " + future.isDone());
```

Результат будет таким:

```
Операция выполняется...
Операция выполняется...
...
Операция была отменена: true
Операция выполнена: true
```

Вот несколько бонусных примеров.

◆ Использование объектов Callable и лямбда-выражений:

```
Future<String> future = executorService.submit(() -> {
    return "Вам привет!";
});
```

◆ Получение объекта Callable, который возвращает null посредством метода

```
Executors.callable(Runnable task):
Callable<Object> callable = Executors.callable(() -> {
    System.out.println("Вам привет!");
});
```

```
Future<Object> future = executorService.submit(callable);
```

◆ Получение объекта Callable, который возвращает результат (T) посредством метода Executors.callable(Runnable task, T result):

```
Callable<String> callable = Executors.callable(() -> {
    System.out.println("Вам привет!");
}, "Hi");
```

```
Future<String> future = executorService.submit(callable);
```

207. Инициирование нескольких операций *Callable*

Поскольку производители (контролеры) не работают одновременно с потребителями (упаковщиками), мы можем просимулировать их работу посредством простого цикла for, который добавляет в очередь 100 проверенных лампочек:

```
private static final BlockingQueue<String> queue
    = new LinkedBlockingQueue<>();
...
private static void simulatingProducers() {
    for (int i = 0; i < MAX_PROD_BULBS; i++) {
        queue.offer("bulb-" + rnd.nextInt(1000));
    }
}
```

Теперь потребители должны упаковать каждую лампочку и вернуть ее. Это означает, что потребитель является объектом Callable:

```
private static class Consumer implements Callable {
    @Override
```

```
public String call() throws InterruptedException {
    String bulb = queue.poll();

    Thread.sleep(100);

    if (bulb != null) {
        logger.info(() -> "Упакована: " + bulb + " потребителем: "
            + Thread.currentThread().getName());
        return bulb;
    }

    return "";
}
```

Но помните, что мы должны предъявлять все операции Callable и ждать до тех пор, пока все они не будут завершены. Этого можно достичь посредством метода `ExecutorService.invokeAll()`. Этот метод берет коллекцию операций (`Collection<? extends Callable<T>>`) и возвращает список экземпляров типа `Future` (`List<Future<T>>`) в качестве аргумента. Любой вызов метода `Future.get()` будет заблокирован до тех пор, пока не будут завершены все экземпляры `Future`.

Таким образом, сначала мы создаем список из 100 операций:

```
private static final Consumer consumer = new Consumer();
...
List<Callable<String>> tasks = new ArrayList<>();
for (int i = 0; i < queue.size(); i++) {
    tasks.add(consumer);
}
```

Далее мы выполняем все эти операции и получаем список экземпляров `Future`:

```
private static ExecutorService consumerService
    = Executors.newWorkStealingPool();
...
List<Future<String>> futures = consumerService.invokeAll(tasks);
```

Наконец, мы обрабатываем (в данном случае показываем) результаты:

```
for (Future<String> future: futures) {
    String bulb = future.get();
    logger.info(() -> "Операция Future завершена: " + bulb);
}
```

Обратите внимание, что первый вызов метода `Future.get()` блокирует до тех пор, пока не будут завершены все экземпляры `Future`. Это приведет к следующему результату:

```
[12:06:41] [INFO] Упакована: bulb-595 потребителем: ForkJoinPool-1-worker-9
...
```

```
[12:06:42] [INFO] Упакована: bulb-478 потребителем: ForkJoinPool-1-worker-15
[12:06:43] [INFO] Операция Future завершена: bulb-595
...
```

Иногда мы хотим предъявить несколько операций и дождаться, пока одна из них не будет завершена. Этого можно достичь посредством метода `ExecutorService.invokeAny()`. По аналогии с методом `invokeAll()`, этот метод получает в качестве аргумента коллекцию операций (`Collection<? extends Callable<T>>`). Но он возвращает результат самой быстрой операции (не операции `Future`) и отменяет все другие операции, которые еще не были завершены. Например:

```
String bulb = consumerService.invokeAny(tasks);
```

Если вы не хотите ждать до тех пор, когда все операции `Future` закончатся, то действуйте следующим образом:

```
int queueSize = queue.size();
List<Future<String>> futures = new ArrayList<>();
for (int i = 0; i < queueSize; i++) {
    futures.add(consumerService.submit(consumer));
}

for (Future<String> future: futures) {
    String bulb = future.get();
    logger.info(() -> "Операция Future завершена: " + bulb);
}
```

Этот код не станет блокировать операции до тех пор, пока все задачи не будут выполнены. Взгляните на пример результата:

```
[12:08:56] [INFO ] Упакована: bulb-894 потребителем: ForkJoinPool-1-worker-7
[12:08:56] [INFO ] Операция Future завершена: bulb-894
[12:08:56] [INFO ] Упакована: bulb-953 потребителем: ForkJoinPool-1-worker-5
...
```

208. Стопоры

Стопор (latch) — это синхронизатор Java, который позволяет одной или несколькими нитями исполнения ждать до тех пор, пока не завершится связка событий в других нитях. Он начинается с заданного счетчика (обычно представляющего число событий, которые следует ожидать), и каждое событие, которое завершается, отвечает за уменьшение этого счетчика. Когда счетчик достигает нуля, все ожидающие нити исполнения могут пройти через него. Это состояние стопора является терминальным. Стопор не может быть сброшен или использован повторно, поэтому ожидаемые события могут произойти только один раз. Схема на рис. 10.13 демонстрирует в четырех шагах принцип работы стопора с тремя нитями исполнения.

В терминах API указанный стопор имплементируется с помощью класса стопора с обратным отсчетом `java.util.concurrent.CountDownLatch`.

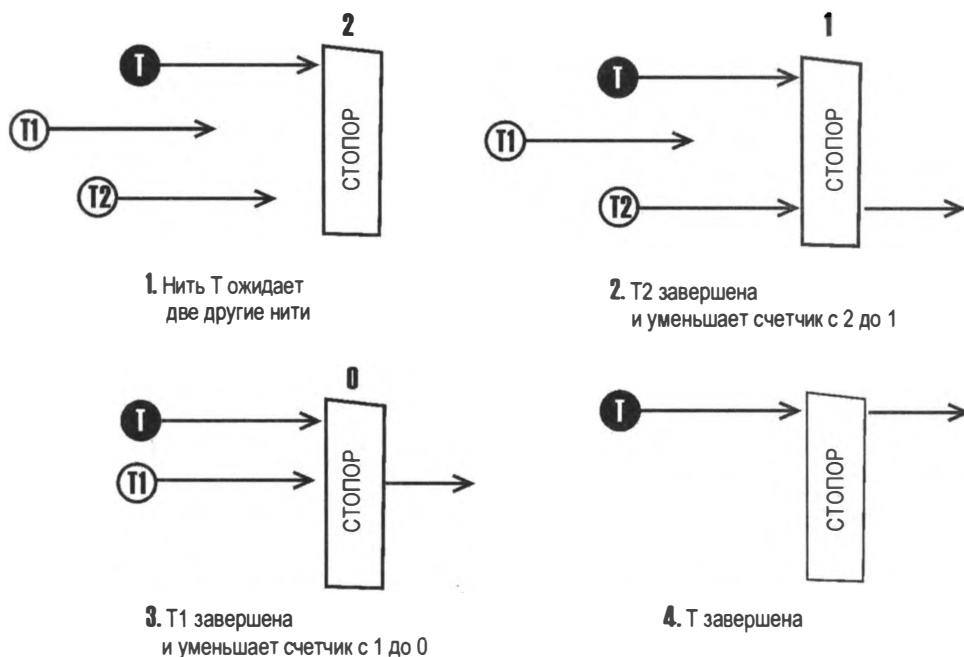


Рис. 10.13

Начальный счетчик задается в конструкторе класса `CountDownLatch` как целое число. Например, `CountDownLatch` со счетчиком, равным 3, можно определить следующим образом:

```
CountDownLatch latch = new CountDownLatch(3);
```

Все нити исполнения, вызывающие метод `await()`, будут заблокированы до тех пор, пока счетчик не достигнет нуля. Таким образом, нить, которая хочет быть заблокированной до тех пор, пока стопор не достигнет терминального состояния, будет вызывать метод `await()`. Каждое событие, которое завершается, может вызвать метод `countDown()`. Этот метод уменьшает счетчик на единицу. Нити, вызвавшие метод `await()`, по-прежнему блокируются до тех пор, пока счетчик не станет равным нулю.

Стопор можно использовать для широкого круга задач. Пока что давайте сосредоточимся на нашей задаче, которая должна просимулировать процесс запуска сервера. Сервер считается запущенным после запуска его внутренних служб. Службы могут запускаться конкурентно и не зависят друг от друга. Процесс запуска сервера занимает некоторое время и требует от нас запуска всех опорных служб этого сервера. Поэтому нить исполнения, которая завершается и проверяет запуск сервера, должна ждать до тех пор, пока все серверные службы (события) не запустятся в других нитях. Если мы допустим, что у нас есть три службы, мы можем написать класс `ServerService` следующим образом:

```
public class ServerInstance implements Runnable {
    private static final Logger logger =
        Logger.getLogger(ServerInstance.class.getName());
```

```
private final CountDownLatch latch = new CountDownLatch(3);

@Override
public void run() {
    logger.info("Сервер готовится к запуску ");
    logger.info("Запуск служб...\n");

    long starting = System.currentTimeMillis();

    Thread service1 = new Thread(new ServerService(latch, "HTTP-слушатели"));
    Thread service2 = new Thread(new ServerService(latch, "JMX"));
    Thread service3 = new Thread(new ServerService(latch, "Коннекторы"));

    service1.start();
    service2.start();
    service3.start();

    try {
        latch.await();
        logger.info(() -> "Сервер успешно запущен за "
            + (System.currentTimeMillis() - starting) / 1000 + " секунд");
    } catch (InterruptedException ex) {
        Thread.currentThread().interrupt();
        // запротоколировать исключение
    }
}
```

Во-первых, мы определяем стопор `CountDownLatch` со счетчиком, равным 3. Во-вторых, мы запускаем службы в трех разных нитях исполнения. Наконец, мы блокируем эту нить посредством метода `await()`. Теперь следующий ниже класс симулирует процесс запуска служб с помощью случайного сна:

```
public class ServerService implements Runnable {
    private static final Logger logger =
        Logger.getLogger(ServerService.class.getName());

    private final String serviceName;
    private final CountDownLatch latch;
    private final Random rnd = new Random();

    public ServerService(CountDownLatch latch, String serviceName) {
        this.latch = latch;
        this.serviceName = serviceName;
    }
}
```

```
@Override
public void run() {
    int startingIn = rnd.nextInt(10) * 1000;

    try {
        logger.info(() -> "Запуск службы '" + serviceName + "' ...");

        Thread.sleep(startingIn);

        logger.info(() -> "Служба '" + serviceName
            + "' успешно запущена за "
            + startingIn / 1000 + " секунд");
    } catch (InterruptedException ex) {
        Thread.currentThread().interrupt();
        // запротokolировать исключение
    } finally {
        latch.countDown();

        logger.info(() -> "Служба '" + serviceName + "' работает...");
    }
}
```

Каждая служба, запущенная успешно (или неуспешно), будет уменьшать стопор посредством метода `countDown()`. Как только счетчик достигает нуля, сервер считается запущенным. Давайте вызовем его:

```
Thread server = new Thread(new ServerInstance());
server.start();
```

Вот возможный результат:

```
[08:49:17] [INFO] Сервер готовится к запуску
[08:49:17] [INFO] Запуск служб...
[08:49:17] [INFO] Запуск службы 'JMX'...
[08:49:17] [INFO] Запуск службы 'Коннекторы'...
[08:49:17] [INFO] Запуск службы 'HTTP-слушатели'...
[08:49:22] [INFO] Служба 'HTTP-слушатели' запущена за 5 секунд
[08:49:22] [INFO] Служба 'HTTP-слушатели' работает...
[08:49:25] [INFO] Служба 'JMX' запущена за 8 секунд
[08:49:25] [INFO] Служба 'JMX' работает...
[08:49:26] [INFO] Служба 'Коннекторы' запущена за 9 секунд
[08:49:26] [INFO] Служба 'Connectors' работает...

[08:49:26] [INFO] Сервер успешно запущен за 9 секунд
```




Во избежание неопределенно долгого ожидания, класс `CountDownLatch` имеет функцию `await()`, которая принимает значение таймаута, `await(long timeout, TimeUnit unit)`. Если время ожидания истекает до того, как счетчик достигает нуля, этот метод возвращает `false`.

209. Барьер

Барьер — это синхронизатор Java, который позволяет группе нитей исполнения (именуемых участниками) достигать общей барьерной точки. В сущности, группа нитей ожидает встретиться друг с другом у барьера. Это похоже на группу друзей, которые выбирают место встречи, и когда все они добираются до этой точки, они идут дальше вместе. Они не будут покидать место встречи до тех пор, пока все не придут на место либо пока не почувствуют, что ждут слишком долго.

Этот синхронизатор хорошо работает для задач, опирающихся на операцию, которую можно разделить на подоперации. Каждая подоперация выполняется в отдельной нити исполнения и ожидает остальные нити. Когда все нити завершены, они объединяют свои результаты в общий результат.

Схема на рис. 10.14 демонстрирует пример барьерного процесса с тремя нитями исполнения.

В терминах API указанный барьер имплементируется с помощью класса циклического барьера `java.util.concurrent.CyclicBarrier`.

Экземпляр класса `CyclicBarrier` можно сконструировать посредством двух конструкторов:

- ♦ один из них дает возможность указывать число участников (это целое число);
- ♦ другой позволяет нам добавлять действие, которое должно произойти после того, как все участники окажутся у барьера (оно является `Runnable`).

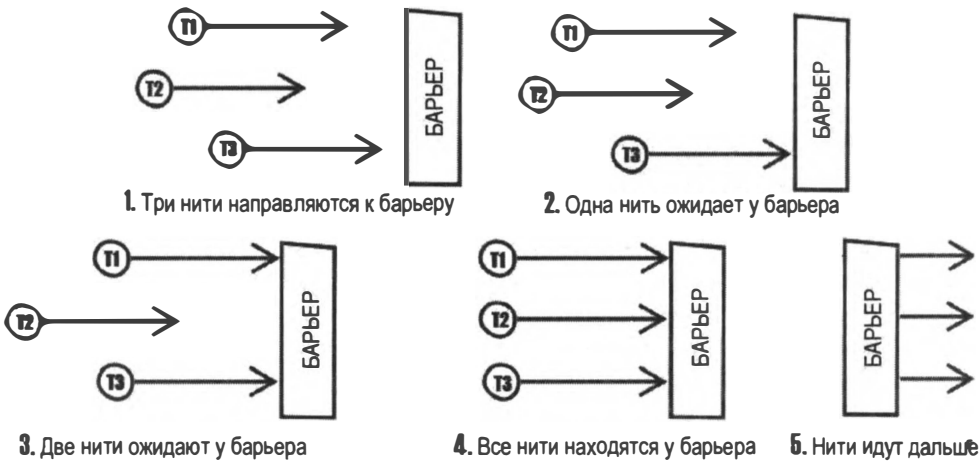


Рис. 10.14

Это действие происходит, когда все нити в участнике прибывают, но до освобождения любых нитей.

Когда нить исполнения готова ждать у барьера, она просто вызывает метод `await()`. Этот метод может ждать бесконечно или в течение указанного таймаута (если время ожидания истекает или нить прерывается, то эта нить освобождается с исключением `TimeoutException`; барьер считается *нарушенным*, и все нити, ожидающие у барьера, освобождаются с исключением `BrokenBarrierException`). Мы можем узнать число участников, требующихся для преодоления этого барьера, посредством метода `getParties()`, а число нитей, в настоящее время ожидающих у барьера, с помощью метода `getNumberWaiting()`.



Метод `await()` возвращает целое число, представляющее индекс прибытия текущей нити исполнения, где индекс `getParties()` — 1 или 0 — указывает соответственно на нить, прибывшую первой или последней.

Допустим, что мы хотим запустить сервер. Он считается запущенным после запуска его внутренних служб. Службы могут быть подготовлены к запуску конкурентно (этот запуск является времязатратным), но они работают взаимозависимо, поэтому, как только они готовы к запуску, они должны быть запущены все сразу.

Таким образом, каждая служба может быть подготовлена к запуску в отдельной нити исполнения. Как только нить будет готова к запуску, она будет ждать у барьера остальные службы. Когда все нити готовы к запуску, они пересекают барьер и начинают работу. Давайте рассмотрим три службы, поэтому `CyclicBarrier` можно определить следующим образом:

```
Runnable barrierAction
    = () -> logger.info("Службы готовы к запуску...");

CyclicBarrier barrier = new CyclicBarrier(3, barrierAction);
```

И давайте подготовим службы посредством трех нитей:

```
public class ServerInstance implements Runnable {
    private static final Logger logger
        = Logger.getLogger(ServerInstance.class.getName());

    private final Runnable barrierAction
        = () -> logger.info("Службы готовы к запуску...");

    private final CyclicBarrier barrier
        = new CyclicBarrier(3, barrierAction);

    @Override
    public void run() {
        logger.info("Сервер готовится к запуску ");
        logger.info("Запуск служб...\n");
    }
}
```

```
long starting = System.currentTimeMillis();

Thread service1 = new Thread( new ServerService(barrier, "HTTP-слушатели"));
Thread service2 = new Thread(new ServerService(barrier, "JMX"));
Thread service3 = new Thread(new ServerService(barrier, "Коннекторы"));

service1.start();
service2.start();
service3.start();

try {
    service1.join();
    service2.join();
    service3.join();

    logger.info(() -> "Сервер успешно запущен за "
        + (System.currentTimeMillis() - starting) / 1000 + " секунд");
} catch (InterruptedException ex) {
    Thread.currentThread().interrupt();
    logger.severe(() -> "Исключение: " + ex);
}
}
```

Класс `ServerService` отвечает за подготовку каждой службы к запуску и блокирование ее у барьера посредством метода `await()`:

```
public class ServerService implements Runnable {
    private static final Logger logger =
        Logger.getLogger(ServerService.class.getName());

    private final String serviceName;
    private final CyclicBarrier barrier;
    private final Random rnd = new Random();

    public ServerService(CyclicBarrier barrier, String serviceName) {
        this.barrier = barrier;
        this.serviceName = serviceName;
    }

    @Override
    public void run() {
        int startingIn = rnd.nextInt(10) * 1000;

        try {
            logger.info(() -> "Подготовка службы '" + serviceName + "' ...");
```

```
Thread.sleep(startingIn);
logger.info(() -> "Служба '" + serviceName
    + "' подготовлена за " + startingIn / 1000
    + " секунд(ы) (ожидает оставшиеся службы)");

barrier.await();

logger.info(() -> "Служба '" + serviceName + "' работает...");
} catch (InterruptedException ex) {
    Thread.currentThread().interrupt();
    logger.severe(() -> "Исключение: " + ex);
} catch (BrokenBarrierException ex) {
    logger.severe(() -> "Исключение... барьер нарушен! " + ex);
}
}
```

А теперь давайте запустим сервер:

```
Thread server = new Thread(new ServerInstance());
server.start();
```

Вот возможный результат (обратите внимание, как нити исполнения были выпущены, чтобы пересечь барьер):

```
[10:38:34] [INFO] Сервер готовится к запуску

[10:38:34] [INFO] Запуск служб...
[10:38:34] [INFO] Подготовка службы 'Коннекторы'...
[10:38:34] [INFO] Подготовка службы 'JMX'...
[10:38:34] [INFO] Подготовка службы 'HTTP-слушатели'...

[10:38:35] [INFO] Служба 'HTTP-слушатели' подготовлена за 1 секунд(ы)
(ожидает оставшиеся службы)
[10:38:36] [INFO] Служба 'JMX' подготовлена за 2 секунд(ы)
(ожидает оставшиеся службы)
[10:38:38] [INFO] Служба 'Коннекторы' подготовлена за 4 секунд(ы)
(ожидает оставшиеся службы)

[10:38:38] [INFO] Службы готовы к запуску...

[10:38:38] [INFO] Служба 'Коннекторы' работает...
[10:38:38] [INFO] Служба 'HTTP-слушатели' работает...
[10:38:38] [INFO] Служба 'JMX' работает...

[10:38:38] [INFO] Сервер успешно запущен за 4 секунды
```



Барьер `CyclicBarrier` является циклическим, потому что он может быть сброшен и использован повторно. Для этого после того, как все нити, ожидающие у барьера, будут освобождены, следует вызвать метод `reset()`, иначе будет выброшено исключение `BrokenBarrierException`.

Барьер, находящийся в *нарушенном состоянии*, приведет к тому, что флаговый метод `isBroken()` вернет значение `true`.

210. Обменник

Обменник (exchanger) — это синхронизатор Java, который позволяет двум нитям исполнения обмениваться объектами в пункте обмена или точке синхронизации.

В общих чертах, этот вид синхронизатора действует как барьер. Две нити ждут друг друга у барьера. Они обмениваются через него объектами и продолжают свои обычные операции, когда оба объекта прибывают.

Схема на рис. 10.15 в четыре шага показывает принцип работы обменника.

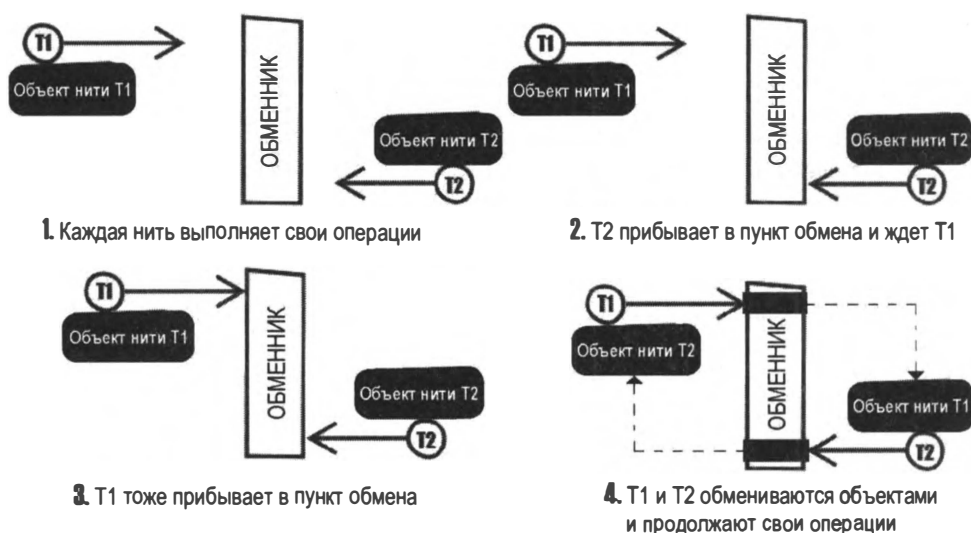


Рис. 10.15

В терминах API указанный синхронизатор представлен классом `java.util.concurrent.Exchanger`. Экземпляр класса `Exchanger` может быть создан посредством пустого конструктора, и указанный класс выставляет наружу два метода `exchange()`:

- ♦ метод, который получает только тот объект, который он предложит;
- ♦ метод, который получает таймаут (если указанное время ожидания истекает перед тем, как еще одна нить входит в обменник, то выбрасывается исключение `TimeoutException`).

Помните нашу сборочную линию для лампочек? Теперь давайте допустим, что производитель (контролер) добавляет проверенные лампочки в корзину (например, список `List<String>`). Когда корзина заполнена, производитель обменивает ее с потребителем (упаковщиком) на пустую корзину (например, еще один список `List<String>`). Указанный процесс повторяется до тех пор, пока работает сборочная линия.

Схема на рис. 10.16 демонстрирует этот процесс.



Рис. 10.16

Таким образом, сначала нам нужен обменник `Exchanger`:

```
private static final int BASKET_CAPACITY = 5;
...
private static final Exchanger<List<String>> exchanger
    = new Exchanger<>();
```

Производитель заполняет корзину и ждет потребителя в пункте обмена:

```
private static final int MAX_PROD_TIME_MS = 2 * 1000;
private static final Random rnd = new Random();
private static volatile boolean runningProducer;
...
private static class Producer implements Runnable {
    private List<String> basket = new ArrayList<>(BASKET_CAPACITY);

    @Override
    public void run() {
        while (runningProducer) {
            try {
                for (int i = 0; i < BASKET_CAPACITY; i++) {
                    String bulb = "bulb-" + rnd.nextInt(1000);
                    Thread.sleep(rnd.nextInt(MAX_PROD_TIME_MS));
                    basket.add(bulb);

                    logger.info(() -> "Проверена и добавлена в корзину: " + bulb);
                }
            }
        }
    }
}
```

```
        logger.info("Производитель: ждет омена корзинами...");

        basket = exchanger.exchange(basket);
    } catch (InterruptedException ex) {
        Thread.currentThread().interrupt();
        logger.severe(() -> "Исключение: " + ex);
        break;
    }
}
}
```

С другой стороны, потребитель ждет в пункте обмена, чтобы получить полную корзину лампочек от производителя, и дает пустую в обмен. Далее, пока производитель снова наполняет корзину, потребитель упаковывает лампочки из полученной корзины. Когда он закончит, то снова пойдет в пункт обмена, чтобы дожидаться еще одной полной корзины. Таким образом, потребитель `Consumer` может быть написан следующим образом:

```
private static final int MAX_CONS_TIME_MS = 5 * 1000;
private static final Random rnd = new Random();
private static volatile boolean runningConsumer;
...
private static class Consumer implements Runnable {
    private List<String> basket = new ArrayList<>(BASKET_CAPACITY);

    @Override
    public void run() {
        while (runningConsumer) {
            try {
                logger.info("Потребитель: ждет обмена корзинами...");
                basket = exchanger.exchange(basket);
                logger.info(() -> "Потребитель: получил следующие лампочки: "
                    + basket);

                for (String bulb: basket) {
                    if (bulb != null) {
                        Thread.sleep(rnd.nextInt(MAX_CONS_TIME_MS));
                        logger.info(() -> "Упакована из корзины: " + bulb);
                    }
                }

                basket.clear();
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

```

        logger.severe(() -> "Исключение: " + ex);
        break;
    }
}
}
}
}

```

Остальная часть исходного кода была опущена для краткости.

Теперь давайте посмотрим на возможный результат:

Запуск сборочной линии...

```

[13:23:13] [INFO] Потребитель: ждет обмена корзинами...
[13:23:15] [INFO] Проверена и добавлена в корзину: bulb-606
...
[13:23:18] [INFO] Производитель: ждет обмена корзинами...
[13:23:18] [INFO] Потребитель: получил следующие лампочки:
[bulb-606, bulb-251, bulb-102, bulb-454, bulb-280]
[13:23:19] [INFO] Проверена и добавлена в корзину: bulb-16
...
[13:23:21] [INFO] Упакована из корзины: bulb-606
...

```

211. Семафоры

Семафор — это синхронизатор Java, позволяющий нам контролировать число нитей исполнения, которые могут обращаться к ресурсу в любой момент времени. В концептуальном плане этот синхронизатор управляет связкой *разрешений* (например, похожих на токены). Нить исполнения, которой требуется доступ к ресурсу, должна приобрести разрешение у синхронизатора. После того как нить завершит свою работу с ресурсом, она должна освободить разрешение, вернув его семафору, чтобы другая нить могла его приобрести. Нить исполнения может приобретать разрешение немедленно (если разрешение является свободным), может ждать определенное количество времени либо может ждать до тех пор, пока разрешение станет свободным. Кроме того, нить исполнения может приобретать и освобождать более одного разрешения за один раз, и нить может освобождать разрешение, даже если она его не приобретала. Благодаря этому разрешение будет добавляться в семафор; поэтому семафор может начинаться с одного числа разрешений и умирать с другим.

В терминах API указанный синхронизатор представлен классом

```
java.util.concurrent.Semaphore.
```

Создать экземпляр класса `Semaphore` можно так же просто, как вызвать один из двух его конструкторов:

```

♦ public Semaphore(int permits);
♦ public Semaphore(int permits, boolean fair).

```


Справедливый семафор гарантирует выдачу оспариваемых разрешений в режиме "первым вошел, первым вышел" (FIFO).

Приобрести разрешение можно посредством метода `acquire()`. Этот процесс может быть представлен следующим перечнем:

- ◆ без аргументов указанный метод будет приобретать разрешение у этого семафора, блокируя до тех пор, пока оно не будет доступно, либо нить не будет прервана;
- ◆ более одного разрешения можно приобрести посредством метода `acquire(int permits)`;
- ◆ попытаться приобрести разрешение и немедленно вернуть значение флага можно посредством методов `tryAcquire()` или `tryAcquire(int permits)`;
- ◆ приобрести разрешение путем ожидания до тех пор, пока оно не станет доступным в течение заданного таймаута (при этом текущая нить не была прервана), можно посредством метода `tryAcquire(int permits, long timeout, TimeUnit unit)`;
- ◆ приобрести разрешение у этого семафора, блокируя выполнение до тех пор, пока оно не будет доступно, можно посредством методов `acquireUninterruptibly()` и `acquireUninterruptibly(int permits)`;
- ◆ для того чтобы освободить разрешение, использовать метод `release()`.

Теперь в нашем сценарии мы будем использовать парикмахерскую, в которой имеются три кресла и которая обслуживает клиентов с дисциплиной доступа FIFO. Клиент пытается занять свое кресло в течение 5 секунд. В конце он освобождает занимаемое кресло. Взгляните на следующий ниже исходный код, чтобы увидеть, как кресло занимает и освобождается:

```
public class Barbershop {
    private static final Logger logger =
        Logger.getLogger(Barbershop.class.getName());

    private final Semaphore seats;

    public Barbershop(int seatsCount) {
        this.seats = new Semaphore(seatsCount, true);
    }

    public boolean acquireSeat(int customerId) {
        logger.info(() -> "Клиент #" + customerId + " пытается занять кресло");

        try {
            boolean acquired = seats.tryAcquire(5 * 1000, TimeUnit.MILLISECONDS);

            if (!acquired) {
                logger.info(() -> "Клиент #" + customerId + " покинул парикмахерскую");

                return false;
            }
        }
    }
}
```

```
logger.info(() -> "Клиент #" + customerId + " занял кресло");

return true;
} catch (InterruptedException ex) {
    Thread.currentThread().interrupt();
    logger.severe(() -> "Исключение: " + ex);
}

return false;
}

public void releaseSeat(int customerId) {
    logger.info(() -> "Клиент #" + customerId + " освободил кресло");
    seats.release();
}
}
```

Если за эти 5 секунд ни одно кресло не было освобождено, то клиент покидает парикмахерскую. Клиент, которому удастся занять кресло, обслуживается парикмахером (это будет затрчено случайное число секунд от 0 до 10). Наконец, клиент освобождает кресло. В исходном коде это можно написать следующим образом:

```
public class BarbershopCustomer implements Runnable {
    private static final Logger logger =
        Logger.getLogger(BarbershopCustomer.class.getName());
    private static final Random rnd = new Random();

    private final Barbershop barbershop;
    private final int customerId;

    public BarbershopCustomer(Barbershop barbershop, int customerId) {
        this.barbershop = barbershop;
        this.customerId = customerId;
    }

    @Override
    public void run() {

        boolean acquired = barbershop.acquireSeat(customerId);
        if (acquired) {
            try {
                Thread.sleep(rnd.nextInt(10 * 1000));
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
                logger.severe(() -> "Исключение: " + ex);
            } finally {
```

```

        barbershop.releaseSeat(customerId);
    }
    } else {
        Thread.currentThread().interrupt();
    }
}
}

```

Давайте приведем 10 клиентов в нашу парикмахерскую:

```
Barbershop bs = new Barbershop(3);
```

```

for (int i = 1; i <= 10; i++) {
    BarbershopCustomer bc = new BarbershopCustomer(bs, i);
    new Thread(bc).start();
}

```

Вот снимок возможного результата:

```

[16:36:17] [INFO] Клиент #10 пытается занять кресло
[16:36:17] [INFO] Клиент #5 пытается занять кресло
[16:36:17] [INFO] Клиент #7 пытается занять кресло
[16:36:17] [INFO] Клиент #5 занял кресло
[16:36:17] [INFO] Клиент #10 занял кресло
[16:36:19] [INFO] Клиент #10 освободил кресло
...

```



Разрешение не приобретается на нитевой основе.

Это означает, что нить T1 может приобрести разрешение у семафора, а нить T2 может его освободить. Разумеется, ответственность за управление этим процессом лежит на разработчике.

212. Фазировщики

Фазировщик (phaser) — это гибкий синхронизатор Java, который сочетает в себе функциональность циклического барьера `CyclicBarrier` и стопора с обратным отсчетом `CountDownLatch` в следующем ниже контексте.

- ◆ Фазировщик состоит из одной или нескольких фаз, которые действуют как барьеры для динамического числа участников (нитей исполнения).
- ◆ В течение срока службы фазировщика число синхронизированных участников (нитей исполнения) может изменяться динамически. Мы можем регистрировать/отменять регистрацию участников.
- ◆ Зарегистрированные в настоящее время участники должны ждать в текущей фазе (барьере), прежде чем переходить к следующему шагу исполнения (следующей фазе), как в случае циклического барьера.
- ◆ Каждая фаза фазировщика может быть идентифицирована посредством ассоциированного числа/индекса, начиная с 0. Первая фаза равна 0, следующая фаза равна 1, потом фаза равна 2, и так до `Integer.MAX_VALUE`.

- ◆ Фазировщик может иметь три типа участников в любой из своих фаз: *зарегистрированные*, *прибывшие* (это зарегистрированные участники, ожидающие в текущей фазе/барьере) и *неприбывшие* (это зарегистрированные участники, которые находятся на пути к текущей фазе).
- ◆ Существуют три типа динамических счетчиков для участников: счетчик для зарегистрированных участников, счетчик для прибывших участников и счетчик для неприбывших участников. Когда все участники прибывают в текущую фазу (число зарегистрированных участников равно числу прибывших участников), фазировщик переходит к следующей фазе.
- ◆ Опционально мы можем исполнить действие (фрагмент кода) непосредственно перед переходом к следующей фазе (когда все участники достигают фазы/барьера).
- ◆ Фазировщик имеет состояние терминации. Число зарегистрированных участников не зависит от терминации, но после терминации все методы синхронизации немедленно возвращаются, не дожидаясь перехода к другой фазе. Схожим образом, попытки зарегистрироваться после терминации не имеют никакого эффекта.

На рис. 10.17 мы видим фазировщика с четырьмя зарегистрированными участниками в фазе 0 и тремя зарегистрированными участниками в фазе 1. У нас также есть несколько разновидностей API, которые обсуждаются далее.

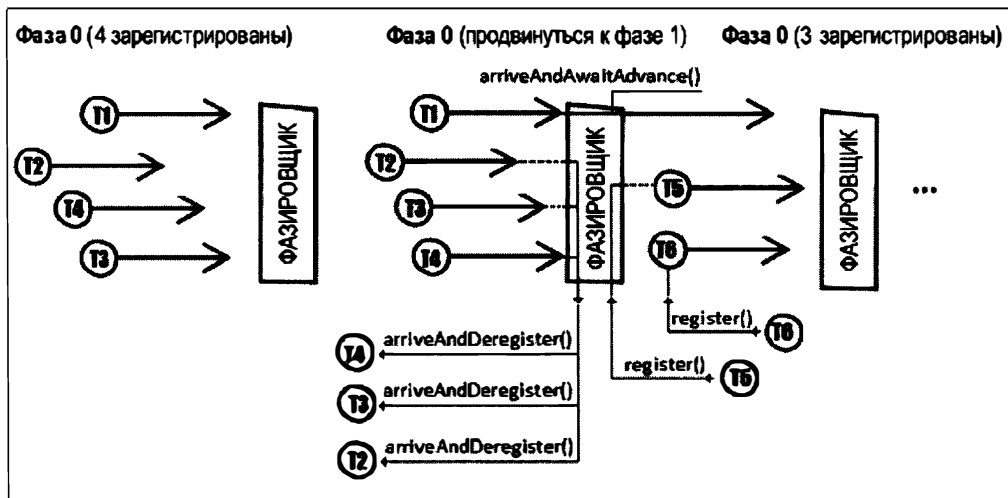


Рис. 10.17



Обычно под участниками мы понимаем нити исполнения (один участник = одна нить исполнения), но фазировщик не ассоциирует участника с конкретной нитью. Фазировщик просто подсчитывает и управляет числом зарегистрированных и deregистрованных участников.

В терминах API указанный синхронизатор представлен классом

`java.util.concurrent.Phaser`.

Экземпляр класса `Phaser` может быть создан с нулем участников, с явным числом участников посредством пустого конструктора либо конструктора, принимающего целочисленный аргумент `Phaser(int parties)`. Фазировщик `Phaser` также может иметь родителя, указываемого посредством `Phaser(Phaser parent)` или `Phaser(Phaser parent)`. Общепринято запускать фазировщика с одним участником, именуемым контролером или контрольным участником. Обычно этот участник живет дольше всего в течение срока службы фазировщика.

Участник может быть зарегистрирован в любое время посредством метода `register()` (на приведенной схеме между фазами 0 и 1 мы регистрируем T5 и T6). Мы также можем зарегистрировать группу участников посредством метода `bulkRegister(int parties)`. Зарегистрированный участник может быть deregистрирован без ожидания других участников посредством метода `arriveAndDeregister()`. Этот метод позволяет участнику прибывать к текущему барьеру (`Phaser`) и deregистрировать его, не дожидаясь прибытия других участников (на рис. 10.17 участники T4, T3 и T2 deregистрируются один за другим). Каждый deregистрированный участник уменьшает число зарегистрированных участников на единицу.

Для того чтобы добраться до текущей фазы (барьера) и дождаться прибытия других участников, нам нужно вызвать метод `arriveAndAwaitAdvance()`. Он блокирует действия до тех пор, пока все зарегистрированные участники не придут в текущую фазу. Все участники продвинулись к следующей фазе этого фазировщика, как только последний зарегистрированный участник придет в текущую фазу.

Опционально, когда все зарегистрированные участники прибывают в текущую фазу, мы можем выполнить конкретные действия путем переопределения метода `onAdvance()` с сигнатурой `boolean onAdvance(int phase, int registeredParties)`. Этот метод возвращает булево значение `true`, если мы хотим инициировать терминацию фазировщика. Дополнено мы можем вынудить терминацию посредством метода `forceTermination()`, и мы можем проверить ее посредством флагового метода `isTerminated()`. Переопределение метода `onAdvance()` требует от нас выполнить расширение класса `Phaser` (обычно посредством анонимного класса).

На данный момент у нас должно быть достаточно деталей, чтобы решить нашу задачу. Итак, мы должны просимулировать процесс запуска сервера в трех фазах объекта `Phaser`. Сервер считается запущенным и работающим после запуска пяти внутренних служб. В первой фазе нам нужно конкурентно запустить три службы. Во второй фазе — конкурентно запустить еще две службы (они могут быть запущены только в том случае, если первые три уже запущены). В третьей фазе сервер выполняет заключительную проверку и считается запущенным и работающим.

Таким образом, нить исполнения (участник), управляющая процессом запуска сервера, можно считать нитью, управляющей остальными нитями (участниками). Это

означает, что мы можем создать объект `Phaser` и зарегистрировать эту управляющую нить (или контроллер) посредством конструктора класса `Phaser`:

```
public class ServerInstance implements Runnable {
    private static final Logger logger =
        Logger.getLogger(ServerInstance.class.getName());

    private final Phaser phaser = new Phaser(1) {
        @Override
        protected boolean onAdvance(int phase, int registeredParties) {
            logger.warning() -> "Фаза:" + phase
                + " Зарегистрированные участники: " + registeredParties);

            return registeredParties == 0;
        }
    };
    ...
}
```

Используя анонимный класс, мы создаем этот объект `Phaser` и переопределяем его метод `onAdvance()`, чтобы определить действие, которое имеет две главные цели:

- ◆ печатать оперативный статус текущей фазы и число зарегистрированных участников;
- ◆ если зарегистрированных участников больше нет, то инициировать терминацию фазировщика.

Этот метод будет использоваться для каждой фазы, когда все зарегистрированные в настоящее время участники придут к текущему барьеру (текущей фазе).

Нити исполнения, управляющие службами сервера, должны запустить эти службы и deregистровать себя в `Phaser`. Таким образом, каждая служба запускается в отдельной нити исполнения, которая deregистрируется в конце своей работы посредством метода `arriveAndDeregister()`. Для этого мы можем использовать следующий класс, имплементирующий интерфейс `Runnable`:

```
public class ServerService implements Runnable {
    private static final Logger logger =
        Logger.getLogger(ServerService.class.getName());

    private final String serviceName;
    private final Phaser phaser;
    private final Random rnd = new Random();

    public ServerService(Phaser phaser, String serviceName) {
        this.phaser = phaser;
        this.serviceName = serviceName;
        this.phaser.register();
    }
}
```

```
@Override
public void run() {
    int startingIn = rnd.nextInt(10) * 1000;

    try {
        logger.info(() -> "Запуск службы '" + serviceName + "' ...");
        Thread.sleep(startingIn);
        logger.info(() -> "Служба '" + serviceName
            + "' была запущена за " + startingIn / 1000
            + " секунд(ы) (ожидает оставшиеся службы)");
    } catch (InterruptedException ex) {
        Thread.currentThread().interrupt();
        logger.severe(() -> "Исключение: " + ex);
    } finally {
        phaser.arriveAndDeregister();
    }
}
```

Теперь управляющая нить исполнения может инициировать процесс запуска для служб service1, service2 и service3. Этот процесс формируется следующим образом:

```
private void startFirstThreeServices() {
    Thread service1 = new Thread(new ServerService(phaser, "HTTP-слушатели"));
    Thread service2 = new Thread(new ServerService(phaser, "JMX"));
    Thread service3 = new Thread(new ServerService(phaser, "Коннекторы"));

    service1.start();
    service2.start();
    service3.start();

    phaser.arriveAndAwaitAdvance(); // фаза 0
}
```

Обратите внимание, что в конце этого метода мы вызываем метод phaser.arriveAndAwaitAdvance(). Это контрольный участник, который ждет прибытия остальных зарегистрированных участников. Остальные зарегистрированные участники (service1, service2 и service3) deregистрируются один за другим до тех пор, пока контрольный участник не останется единственным в Phaser. В этот момент пришло время продвинуться к следующей фазе. Таким образом, контрольный участник является единственным, кто переходит к следующей фазе.

Управляющая нить исполнения может инициировать процесс запуска для служб service4 и service5 схожим с этой имплементацией образом. Этот процесс формируется, как показано ниже:

```
private void startNextTwoServices() {
    Thread service4 = new Thread(
        new ServerService(phaser, "Виртуальные хосты"));
    Thread service5 = new Thread(new ServerService(phaser, "Порты"));
}
```

```

service4.start();
service5.start();

phaser.arriveAndAwaitAdvance(); // фаза 1
}

Наконец, после запуска этих пяти служб управляющая нить выполняет последнюю
сверку, которая была имплементирована в следующем ниже методе как фиктивный
сон Thread.sleep(). Обратите внимание, что в конце этого действия управляющая
нить, запустившая сервер, сама deregистрирует себя в Phaser. Когда это происхо-
дит, это означает, что больше нет зарегистрированных участников, и Phaser терми-
нируется в результате возврата true из метода onAdvance():
private void finalCheckIn() {
    try {
        logger.info("Финализация процесса (должна занять 2 секунды) ...");
        Thread.sleep(2000);
    } catch (InterruptedException ex) {
        Thread.currentThread().interrupt();
        logger.severe(() -> "Исключение: " + ex);
    } finally {
        phaser.arriveAndDeregister(); // фаза 2
    }
}

```

Работа управляющей нити исполнения состоит в том, чтобы вызывать приведенные выше три метода в правильном порядке. Остальная часть кода состоит из несколь-ких операций журналирования/протоколирования, поэтому она была опущена для краткости. Полный исходный код этой задачи прилагается к данной книге.



В любой момент мы можем узнать число зарегистрированных участников по-средством метода `getRegisteredParties()`, число прибывших участников с по-мощью метода `getArrivedParties()` и число неприбывших участников с исполь-зованием метода `getUnarrivedParties()`. Возможно, вы также захотите провер-ить методы `arrive()`, `awaitAdvance(int phase)` и `awaitAdvanceInterruptibly(int phase)`.

Резюме

Настоящая глава очертила основные границы конкурентности (одновременной об-работки) в среде Java и должна была подготовить вас к следующей главе. Мы рас-смотрели несколько фундаментальных задач, связанных с жизненными циклами нитей исполнения, применением замков на уровне объектов и классов, пулами ни-тей исполнения, интерфейсами `Callable` и `Future`.

Вы можете скачать приложения из этой главы, чтобы увидеть результаты и полу-чить дополнительные детали, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

11

Конкурентность — глубокое погружение

Эта глава содержит 13 задач с привлечением конкурентности (псевдоодновременной обработки) в Java, охватывающих такие области, как каркас разветвления/соединения, классы `CompletableFuture`, `ReentrantLock`, `ReentrantReadWriteLock`, `StampedLock`, атомарные переменные, отмена операций, прерываемые методы, переменные типа `ThreadLocal` и тупики. Конкурентность — одна из обязательных тем для любого разработчика, и она никогда не пропускается на собеседованиях при приеме на работу. Вот почему эта и последняя главы так важны.

Прочитав эту главу, вы получите четкое представление о конкурентности, которое необходимо каждому разработчику в среде Java.

Задачи

Используйте следующие задачи для проверки вашего умения программировать конкурентность. Перед тем как обратиться к решениям и скачать примеры программ, настоятельно рекомендуется изучить каждую задачу.

213. **Прерываемые методы.** Написать программу, которая иллюстрирует наилучший подход к работе с прерываемым методом.
214. **Каркас разветвления/соединения.** Написать программу, которая использует каркас разветвления/соединения для суммирования элементов списка. Написать программу, которая использует каркас разветвления/соединения для вычисления числа Фибоначчи в заданной позиции (например, $F_{12} = 144$). В дополнение к этому написать программу, которая иллюстрирует использование класса `CountedCompleter`.
215. **Каркас разветвления/соединения и метод `compareAndSetForkJoinTaskTag()`.** Написать программу, применяющую каркас разветвления/соединения для набора взаимозависимых операций, которые должны быть исполнены только

один раз (например, операция *D* зависит от операции *C* и операции *B*, но операция *C* зависит от операции *B* тоже, и, следовательно, операция *B* должна быть исполнена только один раз, и не два).

216. **Класс `CompletableFuture`.** Написать несколько фрагментов кода для иллюстрации асинхронного кода посредством класса `CompletableFuture`.
217. **Сочетание нескольких объектов класса `CompletableFuture`.** Написать несколько фрагментов кода, иллюстрирующих разные варианты решения задачи сочетания нескольких объектов класса `CompletableFuture` вместе.
218. **Оптимизация занятого ожидания.** Написать эксперимент, иллюстрирующий оптимизацию занятого ожидания посредством метода `onSpinWait()`.
219. **Отмена операции.** Написать эксперимент, иллюстрирующий использование волатильной переменной для удержания состояния отмены процесса.
220. **Переменные `ThreadLocal`.** Написать эксперимент, иллюстрирующий использование переменных `ThreadLocal`.
221. **Атомарные переменные.** Написать программу, которая подсчитывает целые числа от 1 до 1 млн, используя многопоточное приложение (`Runnable`).
222. **Класс `ReentrantLock`.** Написать программу, которая увеличивает целые числа от 1 до 1 млн посредством класса `ReentrantLock`.
223. **Класс `ReentrantReadWriteLock`.** Написать программу, которая симулирует оркестровку процесса чтения-записи посредством класса `ReentrantReadWriteLock`.
224. **Класс `StampedLock`.** Написать программу, которая симулирует оркестровку процесса чтения-записи посредством класса `StampedLock`.
225. **Тупик (обед философов).** Написать программу, которая выявляет и обрабатывает тупиковую ситуацию (круговое ожидание или смертельное объятие), которое может возникать в знаменитой задаче об обеде философов.

Решения

В следующих разделах описаны решения перечисленных выше задач. Помните, что обычно не существует одного единственно правильного варианта решения той или иной задачи. Кроме того, имейте в виду, что приведенные здесь объяснения включают только самые интересные и важные детали, необходимые для решения. Вы можете скачать примеры решений, чтобы увидеть дополнительные детали и поэкспериментировать с программами, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

213. Прерываемые методы

Под прерываемым методом мы подразумеваем блокирующий метод, который может вызвать исключение `InterruptedException`, например `Thread.sleep()`,

`BlockingQueue.take()`, `BlockingQueue.poll(long timeout, TimeUnit unit)` и т. д. Блокирующая нить исполнения обычно находится в состоянии `BLOCKED`, `WAITING` или `TIMED_WAITING`, и если она прерывается, то метод пытается выбросить исключение `InterruptedException` как можно раньше.

Поскольку исключение `InterruptedException` является проверяемым исключением, мы должны его отловить и/или выбросить. Другими словами, если наш метод вызывает метод, который выбрасывает исключение `InterruptedException`, то мы должны быть готовы обработать это исключение. Если мы можем выбросить его (продвинуть исключение вверх в вызывающий код), то это больше не наша работа, и им должен заняться вызывающий код. Поэтому давайте сосредоточимся на том случае, когда мы должны его отловить. Такой случай может возникнуть, когда наш код выполняется внутри объекта `Runnable`, который не может выбрасывать исключение.

Начнем с простого примера. Попытка получить элемент из `BlockingQueue` посредством метода `poll(long timeout, TimeUnit unit)` может быть написана следующим образом:

```
try {
    queue.poll(3000, TimeUnit.MILLISECONDS);
} catch (InterruptedException ex) {
    ...
    logger.info(() -> "Нить прервана? "
        + Thread.currentThread().isInterrupted());
}
```

Попытка опроса элемента из очереди может привести к исключению `InterruptedException`. Нить исполнения может быть прервана в окне продолжительностью 3000 миллисекунд. В случае прерывания (например, `Thread.interrupt()`) мы можем поддаться искушению, думая, что вызов метода `Thread.currentThread().isInterrupted()` в блоке `catch` вернет `true`. В конце концов, мы находимся в блоке `catch`, отлавливающем исключение `InterruptedException`, так что имеет смысл в это поверить. На самом деле он вернет `false`, и ответ находится в исходном коде метода `poll(long timeout, TimeUnit unit)`, приведенном в следующем виде:

```
1: public E poll(long timeout, TimeUnit unit)
    throws InterruptedException {
2:     E e = xfer(null, false, TIMED, unit.toNanos(timeout));
3:     if (e != null || !Thread.interrupted())
4:         return e;
5:     throw new InterruptedException();
6: }
```

Точнее, ответ находится в строке 3. Если нить исполнения была прервана, то метод `Thread.interrupted()` вернет `true` и приведет к строке 5 (`throw new InterruptedException()`). Но помимо проверки, если текущая нить исполнения была прервана, метод `Thread.interrupted()` очищает статус прерванности нити исполне-

ния. Взгляните на приведенную ниже последовательность вызовов для прерванной нити исполнения:

```
Thread.currentThread().isInterrupted(); // true
Thread.interrupted() // true
Thread.currentThread().isInterrupted(); // false
Thread.interrupted() // false
```

Обратите внимание, что вызов `Thread.currentThread().isInterrupted()` проверяет факт прерывания этой нити, не влияя на статус прерванности.

А теперь вернемся к нашему делу. Итак, мы знаем, что нить исполнения была прервана, т. к. мы отловили исключение `InterruptedException`, но статус прерванности был очищен вызовом `Thread.interrupted()`. Это также означает, что код, вызывающий наш код, не будет знать о прерывании.

Наша обязанность — быть хорошими гражданами и восстановить прерывание, вызвав метод `interrupt()`. Благодаря этому код, вызывающий наш код, сможет увидеть, что было выдано прерывание, и действовать соответственно. Правильный код может быть следующим:

```
try {
    queue.poll(3000, TimeUnit.MILLISECONDS);
} catch (InterruptedException ex) {
    ...
    Thread.currentThread().interrupt(); // восстановить прерывание
}
```



В качестве общего правила запомните, что после перехвата исключения `InterruptedException` нужно восстановить прерывание путем вызова метода `Thread.currentThread().interrupt()`.

Давайте рассмотрим задачу, которая высвечивает случай, когда забывают восстановить прерывание. Допустим, что у нас есть операция `Runnable`, которая выполняется до тех пор, пока текущая нить исполнения не будет прервана (например, `while (!Thread.currentThread().isInterrupted()) { ... }`).

На каждой итерации, если статус прерванности текущей нити исполнения равен `false`, мы пытаемся получить элемент из очереди `BlockingQueue`.

Следующий код является имплементацией:

```
Thread thread = new Thread(() -> {
    // некая фиктивная очередь
    TransferQueue<String> queue = new LinkedTransferQueue<>();
    while (!Thread.currentThread().isInterrupted()) {
        try {
            logger.info(() -> "В течение 3 секунд нить "
                + Thread.currentThread().getName()
                + " будет пытаться извлечь элемент из очереди...");
            queue.poll(3000, TimeUnit.MILLISECONDS);
        }
```

```
} catch (InterruptedException ex) {  
    logger.severe(() -> "InterruptedException! Нить "  
        + Thread.currentThread().getName() + " была прервана!");  
    Thread.currentThread().interrupt();  
}  
  
logger.info(() -> "Исполнение было остановлено!");  
});
```

В качестве вызывающего кода (еще одной нити исполнения) мы запускаем вышеупомянутую нить, погружаемся в сон на 1,5 секунды, только чтобы дать время этой нити исполнения войти в метод `poll()`, а затем прерываем ее. Это показано в следующем фрагменте кода:

```
thread.start();  
Thread.sleep(1500);  
thread.interrupt();
```

Это приведет к исключению `InterruptedException`.

Исключение регистрируется, и прерывание восстанавливается.

На следующем шаге цикл `while` вычисляет метод

`Thread.currentThread().isInterrupted()`, получая `false`, и выходит.

В итоге результат будет таким:

```
[18:02:43] [INFO] В течение 3 секунд нить Thread-0  
                будет пытаться извлечь элемент из очереди...  
[18:02:44] [SEVERE] InterruptedException!  
                Нить Thread-0 была прервана!  
[18:02:45] [INFO] Исполнение было остановлено!
```

Теперь давайте прокомментируем строку кода, которая восстанавливает прерывание:

```
...  
} catch (InterruptedException ex) {  
    logger.severe(() -> "InterruptedException! Нить "  
        + Thread.currentThread().getName() + " была прервана!");  
  
    // обратите внимание, что строка ниже закомментирована  
    // Thread.currentThread().interrupt();  
}  
...
```

На этот раз блок `while` будет работать вечно, т. к. его пропускное условие всегда оценивается как `true`.

Этот код не сможет действовать по прерыванию, поэтому результат будет следующим:

```
[18:05:47] [INFO] В течение 3 секунд нить Thread-0  
                будет пытаться извлечь элемент из очереди...
```

```
[18:05:48] [SEVERE] InterruptedException!
           Нить Thread-0 была прервана!
[18:05:48] [INFO] В течение 3 секунд нить Thread-0
           будет пытаться извлечь элемент из очереди...
...
```



В качестве общего правила запомните, что единственный приемлемый случай, когда мы можем проглотить прерывание (не восстанавливать прерывание), — это ситуация, когда мы можем управлять всем стеком вызовов (например, `extend Thread`).

В противном случае отлавливание исключения `InterruptedException` также должно содержать вызов метода `Thread.currentThread().interrupt()`.

214. Каркас разветвления/соединения

Мы уже познакомились с каркасом разветвления/соединения в *разд. "Пул обкрадывающих нитей исполнения"*.

В общих чертах каркас разветвления/соединения предназначен для того, чтобы брать большую операцию (в типичной ситуации под большой мы понимаем крупный объем данных) и рекурсивно делить ее на более мелкие операции (подоперации), которые могут выполняться в параллельном режиме. В конце, после выполнения всех подопераций, их результаты соединяются (объединяются) в один результат.

Схема на рис. 11.1 является визуальным представлением принципа работы разветвления/соединения.

В терминах API разветвление/соединение может быть создано посредством класса `java.util.concurrent.ForkJoinPool`.

До JDK 8 рекомендуемый подход опирался на публичную статическую переменную следующим образом:

```
public static ForkJoinPool forkJoinPool = new ForkJoinPool();
```

Начиная с JDK 8, мы можем сделать это так:

```
ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();
```

Оба подхода позволяют избежать неприятной ситуации наличия слишком большого числа пулов нитей исполнения на одной JVM, обусловленного параллельными операциями, которые создали свои пулы.



Для создания собственного экземпляра класса `ForkJoinPool` опирайтесь на конструкторы этого класса. JDK 9 добавил наиболее полный из них (подробности см. в документации).

Объект `ForkJoinPool` управляет операциями. Базовым типом исполняемой операции в объекте `ForkJoinPool` является тип `ForkJoinTask<V>`.

Точнее, исполняются следующие операции:

- ◆ `RecursiveAction` для операций `void`;
- ◆ `RecursiveTask<V>` для операций, возвращающих значение;
- ◆ `CountedCompleter<T>` для операций, которым необходимо запоминать число операций, ожидающих исполнения.

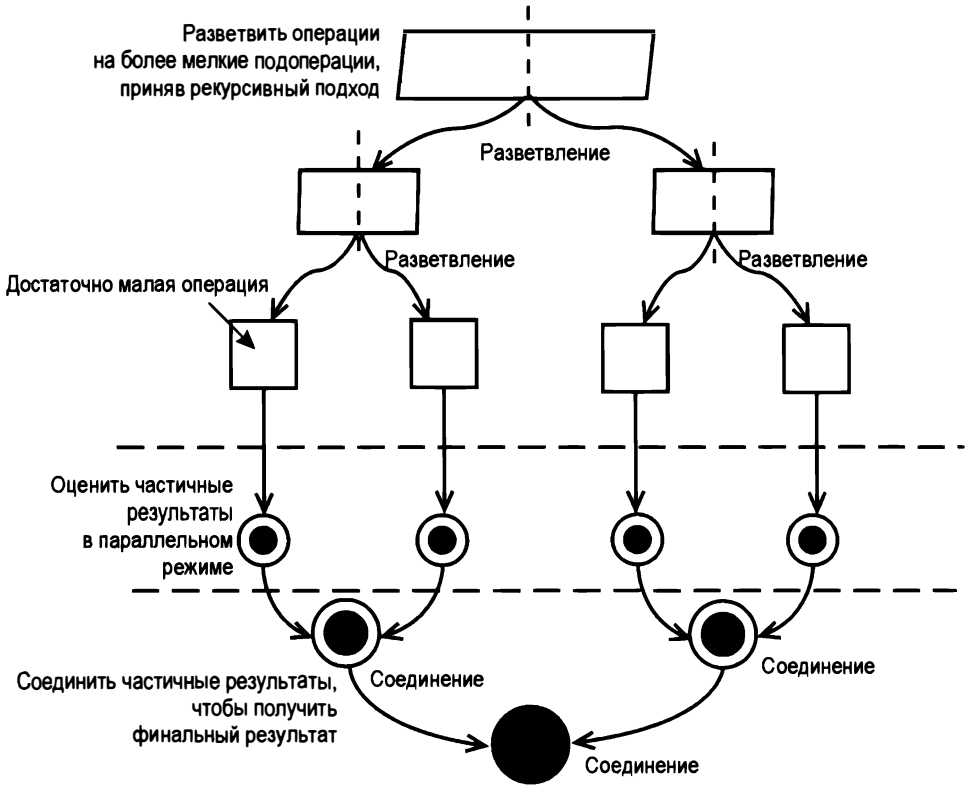


Рис. 11.1

Все три типа операций имеют абстрактный метод с именем `compute()`, в котором формируется логика задачи.

Предъявить операции в пул `ForkJoinPool` можно посредством следующих методов:

- ◆ `execute()` и `submit()`;
- ◆ `invoke()` для разветвления операции и ожидания результата;
- ◆ `invokeAll()` для разветвления связки операций (например, коллекции);
- ◆ `fork()` для организации асинхронного исполнения этой операции в пуле и `join()` для возврата результата вычисления, когда он будет готов.

Начнем с задачи, решаемой посредством класса `RecursiveTask`.

Вычисление суммы посредством класса *RecursiveTask*

Для того чтобы продемонстрировать разветвляющее поведение каркаса, допустим, что у нас есть список чисел и мы хотим вычислить сумму этих чисел. Для этого мы рекурсивно разбиваем (разветвляем) этот список до тех пор, пока он не будет больше заданного порога `THRESHOLD`, используя метод `createSubtasks()`. Каждая операция добавляется в список `List<SumRecursiveTask>`. В конце этот список предьявляется в объект `ForkJoinPool` посредством метода `invokeAll(Collection<T> tasks)`. Это делается с помощью следующего кода:

```
public class SumRecursiveTask extends RecursiveTask<Integer> {
    private static final Logger logger
        = Logger.getLogger(SumRecursiveTask.class.getName());

    private static final int THRESHOLD = 10;

    private final List<Integer> worklist;

    public SumRecursiveTask(List<Integer> worklist) {
        this.worklist = worklist;
    }

    @Override
    protected Integer compute() {
        if (worklist.size() <= THRESHOLD) {
            return partialSum(worklist);
        }

        return ForkJoinTask.invokeAll(createSubtasks())
            .stream()
            .mapToInt(ForkJoinTask::join)
            .sum();
    }

    private List<SumRecursiveTask> createSubtasks() {
        List<SumRecursiveTask> subtasks = new ArrayList<>();
        int size = worklist.size();

        List<Integer> worklistLeft = worklist.subList(0, (size + 1) / 2);
        List<Integer> worklistRight = worklist.subList((size + 1) / 2, size);

        subtasks.add(new SumRecursiveTask(worklistLeft));
        subtasks.add(new SumRecursiveTask(worklistRight));

        return subtasks;
    }
}
```



```
private Integer partialSum(List<Integer> worklist) {
    int sum = worklist.stream()
        .mapToInt(e -> e)
        .sum();

    logger.info(() -> "Частичная сумма: " + worklist + " = "
        + sum + "\tНить: " + Thread.currentThread().getName());

    return sum;
}
}
```

Для проверки этой имплементации нам нужны список и объект `ForkJoinPool`, как показано ниже:

```
ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();

Random rnd = new Random();
List<Integer> list = new ArrayList<>();

for (int i = 0; i < 200; i++) {
    list.add(1 + rnd.nextInt(10));
}

SumRecursiveTask sumRecursiveTask = new SumRecursiveTask(list);
Integer sumAll = forkJoinPool.invoke(sumRecursiveTask);
```

```
logger.info(() -> "Окончательная сумма: " + sumAll);
```

Возможный результат будет следующим:

```
...
[15:17:06] Частичная сумма: [1, 3, 6, 6, 2, 5, 9] = 32
ForkJoinPool.commonPool-worker-9
...
[15:17:06] Частичная сумма: [1, 9, 9, 8, 9, 5] = 41
ForkJoinPool.commonPool-worker-7
[15:17:06] Окончательная сумма: 1084
```

Вычисление чисел Фибоначчи посредством класса *RecursiveAction*

Обычно обозначаемые как F_n , числа Фибоначчи представляют собой последовательность, которая соответствует следующей формуле:

$$F_0 = 0, F_1 = 1, \dots, F_n = F_{n-1} + F_{n-2} \ (n > 1).$$

Ниже приведен фрагмент последовательности чисел Фибоначчи:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Имплементировать числа Фибоначчи посредством объекта RecursiveAction можно так:

```
public class FibonacciRecursiveAction extends RecursiveAction {
    private static final Logger logger =
        Logger.getLogger(FibonacciRecursiveAction.class.getName());
    private static final long THRESHOLD = 5;

    private long nr;

    public FibonacciRecursiveAction(long nr) {
        this.nr = nr;
    }

    @Override
    protected void compute() {
        final long n = nr;

        if (n <= THRESHOLD) {
            nr = fibonacci(n);
        } else {
            nr = ForkJoinTask.invokeAll(createSubtasks(n))
                .stream()
                .mapToLong(x -> x.fibonacciNumber())
                .sum();
        }
    }

    private List<FibonacciRecursiveAction> createSubtasks(long n) {
        List<FibonacciRecursiveAction> subtasks = new ArrayList<>();

        FibonacciRecursiveAction fibonacciMinusOne
            = new FibonacciRecursiveAction(n - 1);
        FibonacciRecursiveAction fibonacciMinusTwo
            = new FibonacciRecursiveAction(n - 2);

        subtasks.add(fibonacciMinusOne);
        subtasks.add(fibonacciMinusTwo);

        return subtasks;
    }

    private long fibonacci(long n) {
        logger.info(() -> "Число: " + n
            + " Нить: " + Thread.currentThread().getName());

        if (n <= 1) {
            return n;
        }
    }
}
```

```
        return fibonacci(n - 1) + fibonacci(n - 2);
    }

    public long fibonacciNumber() {
        return nr;
    }
}
```

Для испытания этой имплементации нам понадобится объект `ForkJoinPool`:

```
ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();
```

```
FibonacciRecursiveAction fibonacciRecursiveAction
    = new FibonacciRecursiveAction(12);
forkJoinPool.invoke(fibonacciRecursiveAction);
```

```
logger.info(() -> "Фибоначчи: " + fibonacciRecursiveAction.fibonacciNumber());
```

Результат для F_{12} выглядит следующим образом:

```
[15:40:46] Число: 5 Нить: ForkJoinPool.commonPool-worker-3
[15:40:46] Число: 5 Нить: ForkJoinPool.commonPool-worker-13
[15:40:46] Число: 4 Нить: ForkJoinPool.commonPool-worker-3
[15:40:46] Число: 4 Нить: ForkJoinPool.commonPool-worker-9
...
[15:40:49] Число: 0 Нить: ForkJoinPool.commonPool-worker-7
[15:40:49] Фибоначчи: 144
```

Использование класса *CountedCompleter*

Класс `CountedCompleter` был добавлен в JDK 8 как тип `ForkJoinTask`.

Работа класса `CountedCompleter` состоит в том, чтобы запомнить число операций, ожидающих исполнения (не меньше и не больше). Мы можем устанавливать счетчик ожиданий посредством метода `setPendingCount()` либо увеличивать его с помощью явной дельты с помощью метода `addToPendingCount(int delta)`. Обычно мы вызываем эти методы непосредственно перед разветвлением (например, если мы разветвляем дважды, то вызываем `addToPendingCount(2)` или `setPendingCount(2)` в зависимости от случая).

В методе `compute()` мы уменьшаем число ожиданий посредством методов `tryComplete()` либо `propagateCompletion()`. При вызове метода `tryComplete()` с счетчиком ожиданий, равным нулю, или вызове метода безусловного завершения `complete()` вызывается метод `onCompletion()`. Метод `propagateCompletion()` аналогичен методу `tryComplete()`, но он не вызывает `onCompletion()`.

Объект `CountedCompleter` может дополнительно возвращать вычисленное значение. Для этого мы должны переопределить метод `getRawResult()`, чтобы вернуть значение.

Следующий ниже код суммирует все значения списка посредством объекта класса CountedCompleter:

```
public class SumCountedCompleter extends CountedCompleter<Long> {
    private static final Logger logger
        = Logger.getLogger(SumCountedCompleter.class.getName());
    private static final int THRESHOLD = 10;
    private static final LongAdder sumAll = new LongAdder();

    private final List<Integer> worklist;

    public SumCountedCompleter(
        CountedCompleter<Long> c, List<Integer> worklist) {
        super(c);
        this.worklist = worklist;
    }

    @Override
    public void compute() {
        if (worklist.size() <= THRESHOLD) {
            partialSum(worklist);
        } else {
            int size = worklist.size();

            List<Integer> worklistLeft = worklist.subList(0, (size + 1) / 2);
            List<Integer> worklistRight = worklist.subList((size + 1) / 2, size);

            addToPendingCount(2);
            SumCountedCompleter leftTask
                = new SumCountedCompleter(this, worklistLeft);
            SumCountedCompleter rightTask
                = new SumCountedCompleter(this, worklistRight);

            leftTask.fork();
            rightTask.fork();
        }

        tryComplete();
    }

    @Override
    public void onCompletion(CountedCompleter<?> caller) {
        logger.info(() -> "Нить завершена: "
            + Thread.currentThread().getName());
    }
}
```

```
@Override
public Long getRawResult() {
    return sumAll.sum();
}

private Integer partialSum(List<Integer> worklist) {
    int sum = worklist.stream()
        .mapToInt(e -> e)
        .sum();

    sumAll.add(sum);

    logger.info(() -> "Частичная сумма: " + worklist + " = "
        + sum + "\tНить: " + Thread.currentThread().getName());

    return sum;
}
}
```

Теперь давайте посмотрим на потенциальный вызов и результат:

```
ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();
Random rnd = new Random();
List<Integer> list = new ArrayList<>();

for (int i = 0; i < 200; i++) {
    list.add(1 + rnd.nextInt(10));
}

SumCountedCompleter sumCountedCompleter
    = new SumCountedCompleter(null, list);
forkJoinPool.invoke(sumCountedCompleter);
```

```
logger.info(() -> "Готово! Результат: "
    + sumCountedCompleter.getRawResult());
```

Результат будет следующим:

```
[11:11:07] Частичная сумма: [7, 7, 8, 5, 6, 10] = 43
    ForkJoinPool.commonPool-worker-7
[11:11:07] Частичная сумма: [9, 1, 1, 6, 1, 2] = 20
    ForkJoinPool.commonPool-worker-3
...
[11:11:07] Нить завершена: ForkJoinPool.commonPool-worker-15
[11:11:07] Готово! Результат: 1159
```

215. Каркас разветвления/соединения метод *compareAndSetForkJoinTaskTag()*

Теперь, когда мы знакомы с каркасом разветвления/соединения, давайте рассмотрим еще одну задачу. На этот раз допустим, что у нас есть набор объектов `ForkJoinTask`, которые являются взаимозависимыми. Схема на рис. 11.2 может рассматриваться как вариант использования.

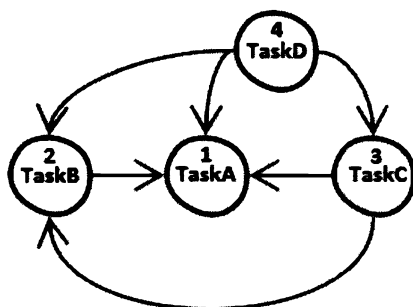


Рис. 11.2

Вот описание приведенной схемы:

- ◆ операция `TaskD` имеет три зависимости: `TaskA`, `TaskB` и `TaskC`;
- ◆ операция `TaskC` имеет две зависимости: `TaskA` и `TaskB`;
- ◆ операция `TaskB` имеет одну зависимость: `TaskA`;
- ◆ операция `TaskA` не имеет зависимостей.

В исходном коде мы сформируем ее следующим образом:

```

ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();

Task taskA = new Task("Task-A", new Adder(1));
Task taskB = new Task("Task-B", new Adder(2), taskA);
Task taskC = new Task("Task-C", new Adder(3), taskA, taskB);
Task taskD = new Task("Task-D", new Adder(4), taskA, taskB, taskC);

forkJoinPool.invoke(taskD);
  
```

Объект `Adder` — это простой объект `Callable`, который должен выполняться только один раз для каждой операции (т. е. один раз для операций `TaskD`, `TaskC`, `TaskB` и `TaskA`). Объект `Adder` иницируется в следующем ниже коде:

```

private static class Adder implements Callable {
    private static final AtomicInteger result = new AtomicInteger();

    private Integer nr;
  
```

```
public Adder(Integer nr) {
    this.nr = nr;
}

@Override
public Integer call() {
    logger.info(() -> "Добавление числа: " + nr
        + " нитью:" + Thread.currentThread().getName());

    return result.addAndGet(nr);
}
}
```

Мы уже знаем, как использовать каркас разветвления/соединения для операций с ациклическими и/или неповторяемыми (или нам все равно, повторяются они или нет) зависимостями завершения. Но если мы имплементируем его таким образом, то объект `Callable` будет вызываться более одного раза для каждой операции. Например, операция `TaskA` появляется как зависимость для трех других операций, поэтому `Callable` будет вызываться трижды. Мы хотим ее вызывать только один раз.

Очень удобное средство класса `ForkJoinPool`, добавленное в JDK 8, состоит в атомарном тегировании значением типа `short`:

- ◆ метод `short getForkJoinTaskTag()` возвращает тег для этой операции;
- ◆ метод `short setForkJoinTaskTag(short newValue)` атомарно устанавливает теговое значение для этой операции и возвращает старое значение;
- ◆ метод `boolean compareAndSetForkJoinTaskTag(short expect, short update)` возвращает `true`, если текущее значение было равно `expect` и было изменено на `update`.

Другими словами, метод `compareAndSetForkJoinTaskTag()` позволяет пометить операцию тегом как посещенную `VISITED`. После того как операция помечена в качестве посещенной, она не будет исполнена. Давайте посмотрим на это в следующих строках кода:

```
public class Task<Integer> extends RecursiveTask<Integer> {
    private static final Logger logger = Logger.getLogger(Task.class.getName());
    private static final short UNVISITED = 0;
    private static final short VISITED = 1;

    private Set<Task<Integer>> dependencies = new HashSet<>();

    private final String name;
    private final Callable<Integer> callable;

    public Task(String name, Callable<Integer> callable,
        Task<Integer> ...dependencies) {
```

```

    this.name = name;
    this.callable = callable;
    this.dependencies = Set.of(dependencies);
}

@Override
protected Integer compute() {
    dependencies.stream()
        .filter((task) -> (task.updateTaskAsVisited()))
        .forEachOrdered((task) -> {
            logger.info(() -> "Помечена: " + task + "("
                + task.getForkJoinTaskTag() + ")");

            task.fork();
        });

    for (Task task: dependencies) {
        task.join();
    }

    try {
        return callable.call();
    } catch (Exception ex) {
        logger.severe(() -> "Исключение: " + ex);
    }

    return null;
}

public boolean updateTaskAsVisited() {
    return compareAndSetForkJoinTaskTag(UNVISITED, VISITED);
}

@Override
public String toString() {
    return name + " | dependencies=" + dependencies + " ";
}
}

```

И вероятный результат может выглядеть следующим образом:

```

[10:30:53] [INFO] Помечена: Task-B(1)
[10:30:53] [INFO] Помечена: Task-C(1)
[10:30:53] [INFO] Помечена: Task-A(1)
[10:30:53] [INFO] Добавление числа: 1
                    нитью:ForkJoinPool.commonPool-worker-3

```



```
[10:30:53] [INFO] Добавление числа: 2
                  нитью:ForkJoinPool.commonPool-worker-3
[10:30:53] [INFO] Добавление числа: 3
                  нитью:ForkJoinPool.commonPool-worker-5
[10:30:53] [INFO] Добавление числа: 4
                  нитью:main
[10:30:53] [INFO] Результат: 10
```

216. Класс *CompletableFuture*

JDK 8 сделал значительный шаг вперед в мире асинхронного программирования, усилив класс `Future` классом `CompletableFuture`. Главные ограничения класса `Future`:

- ◆ он не может быть явно завершен;
- ◆ он не поддерживает обратные вызовы для выполнения действий на результате;
- ◆ обратные вызовы не могут выстраиваться в цепочку или объединяться для получения многосложных асинхронных конвейеров;
- ◆ он не обеспечивает обработку исключений.

У класса `CompletableFuture` этих ограничений нет. Простой, но бесполезный `CompletableFuture` можно написать следующим образом:

```
CompletableFuture<Integer> completableFuture = new CompletableFuture<>();
```

Результат может быть получен посредством блокирующего метода `get()`:

```
completableFuture.get();
```

В дополнение к этому рассмотрим несколько примеров работы асинхронных операций в контексте платформы электронной коммерции. Мы добавляем эти примеры во вспомогательный класс `CustomerAsyncns`.

Выполнение асинхронных операций и возврат *void*

Задача пользователя: печатать тот или иной заказ клиента.

Поскольку печать — это процесс, который не должен возвращать результат, такая работа как раз подходит для метода `runAsync()`. Он может выполнять операцию асинхронно и не возвращает результат. Другими словами, он берет объект `Runnable` и возвращает `CompletableFuture<Void>`; это показано в следующем фрагменте кода:

```
public static void printOrder() {
    CompletableFuture<Void> cfPrintOrder
        = CompletableFuture.runAsync(new Runnable() {

        @Override
        public void run() {
            logger.info(() -> "Заказ печатается: "
                + Thread.currentThread().getName());
        }
    });
}
```

```
        Thread.sleep(500);
    }
});

cfPrintOrder.get(); // блокировать до тех пор, пока заказ не будет напечатан
logger.info("Заказ клиента напечатан...\n");
}
```

Как вариант, мы можем написать его с помощью лямбда-выражения:

```
public static void printOrder() {
    CompletableFuture<Void> cfPrintOrder
        = CompletableFuture.runAsync(() -> {

        logger.info(() -> "Заказ печатается: "
            + Thread.currentThread().getName());
        Thread.sleep(500);
    });

    cfPrintOrder.get(); // блокировать до тех пор, пока заказ не будет напечатан
    logger.info("Заказ клиента напечатан...\n");
}
```

Выполнение асинхронной операции и возврат ее результата

Задача пользователя: получить сводку заказа того или иного клиента.

На этот раз асинхронная операция должна возвращать результат, и поэтому метод `runAsync()` бесполезен. Эта операция подходит для метода `supplyAsync()`. Он берет поставщика `Supplier<T>` и возвращает `CompletableFuture<T>`. `T` — это тип результата, получаемого от данного поставщика посредством метода `get()`. В исходном коде мы можем решить эту задачу следующим образом:

```
public static void fetchOrderSummary() {
    CompletableFuture<String> cfOrderSummary
        = CompletableFuture.supplyAsync(() -> {

        logger.info(() -> "Получить сводку заказа: "
            + Thread.currentThread().getName());
        Thread.sleep(500);

        return "Сводка заказа #93443";
    });

    // ждать до тех пор, пока не появится сводка; эта операция блокирует
    String summary = cfOrderSummary.get();
    logger.info(() -> "Сводка заказа: " + summary + "\n");
}
```

Выполнение асинхронной операции и возврат результата посредством явного пула нитей исполнения

Задача пользователя: получить сводку заказа того или иного клиента.

По умолчанию, как и в предыдущих примерах, асинхронные операции выполняются в нитях исполнения, полученных из глобального пула `ForkJoinPool.commonPool()`. С помощью журнального метода `Thread.currentThread().getName()` мы видим что-то вроде `ForkJoinPool.commonPool-worker-3`.

Но мы также можем использовать явный собственный пул потоков `Executor`. Все методы `CompletableFuture`, которые способны выполнять асинхронные операции, обеспечивают разновидность, которая принимает `Executor`.

Вот пример использования одного-единственного пула нитей исполнения:

```
public static void fetchOrderSummaryExecutor() {
    ExecutorService executor = Executors.newSingleThreadExecutor();

    CompletableFuture<String> cfOrderSummary
        = CompletableFuture.supplyAsync(() -> {
            logger.info(() -> "Получить сводку заказа: "
                + Thread.currentThread().getName());
            Thread.sleep(500);

            return "Сводка заказа #91022";
        }, executor);

    // ждать до тех пор, пока не появится сводка; эта операция блокирует
    String summary = cfOrderSummary.get();
    logger.info(() -> "Сводка заказа: " + summary + "\n");
    executor.shutdownNow();
}
```

Прикрепление функции обратного вызова, которая обрабатывает результат асинхронной операции и возвращает результат

Задача пользователя: получить счет-фактуру заказа того или иного клиента, а затем вычислить ее итоговую сумму и подписать ее.

В таких задачах опираться на блокирующий метод `get()` не очень полезно. Нам нужен метод обратного вызова, который будет вызываться автоматически, когда результат `CompletableFuture` будет доступен.

Таким образом, мы не хотим ждать результата. Когда счет-фактура готова (это результат `CompletableFuture`), метод обратного вызова должен вычислить итоговое значение, а затем другой обратный вызов должен подписать счет-фактуру. Это можно достичь посредством метода `thenApply()`.

Метод `thenApply()` полезен для обработки и преобразования результата `CompletableFuture`, когда тот прибывает. Он берет `Function<T, R>` в качестве аргумента. Давайте посмотрим, как он работает:

```
public static void fetchInvoiceTotalSign() {
    CompletableFuture<String> cfFetchInvoice
        = CompletableFuture.supplyAsync(() -> {

        logger.info(() -> "Получить счет-фактуру: "
            + Thread.currentThread().getName());
        Thread.sleep(500);

        return "Счет-фактура #3344";
    });

    CompletableFuture<String> cfTotalSign = cfFetchInvoice
        .thenApply(o -> o + " Итого: $145")
        .thenApply(o -> o + " Подписано");

    String result = cfTotalSign.get();
    logger.info(() -> "Счет-фактура: " + result + "\n");
}
```

Как вариант, мы можем выстроить цепочку следующим образом:

```
public static void fetchInvoiceTotalSign() {
    CompletableFuture<String> cfTotalSign
        = CompletableFuture.supplyAsync(() -> {

        logger.info(() -> "Получить счет-фактуру: "
            + Thread.currentThread().getName());
        Thread.sleep(500);

        return "Счет-фактура #3344";
    }).thenApply(o -> o + " Итого: $145")
        .thenApply(o -> o + " Подписано");

    String result = cfTotalSign.get();
    logger.info(() -> "Счет-фактура: " + result + "\n");
}
```



Проверьте также методы `applyToEither()` и `applyToEitherAsync()`. Когда либо этот, либо другой данный этап завершается нормально, оба метода возвращают новый этап завершения, исполняемый предоставленной функцией, в которой результат выступает в качестве аргумента.

Прикрепление функции обратного вызова, которая обрабатывает результат асинхронной операции и возвращает *void*

Задача пользователя: получить заказ того или иного клиента и напечатать его.

В типичной ситуации обратный вызов, который не возвращает результат, действует как терминальное действие асинхронного конвейера.

Это поведение может быть получено посредством метода `thenAccept()`. Он берет `Consumer<T>` и возвращает `CompletableFuture<Void>`. Данный метод может обрабатывать и трансформировать результат `CompletableFuture`, но не возвращает результат. Таким образом, он может принять заказ, который является результатом `CompletableFuture`, и напечатать его, как показано в следующем фрагменте кода:

```
public static void fetchAndPrintOrder() {
    CompletableFuture<String> cfFetchOrder
        = CompletableFuture.supplyAsync(() -> {

        logger.info(() -> "Получить заказ: " + Thread.currentThread().getName());
        Thread.sleep(500);

        return "Заказ #1024";
    });

    CompletableFuture<Void> cfPrintOrder = cfFetchOrder.thenAccept(
        o -> logger.info(() -> "Печать заказа " + o +
            " нитью: " + Thread.currentThread().getName()));

    cfPrintOrder.get();
    logger.info("Заказ получен и напечатан \n");
}
```

Как вариант, код можно записать компактнее следующим образом:

```
public static void fetchAndPrintOrder() {
    CompletableFuture<Void> cfFetchAndPrintOrder
        = CompletableFuture.supplyAsync(() -> {

        logger.info(() -> "Получить заказ: " + Thread.currentThread().getName());
        Thread.sleep(500);

        return "Заказ #1024";
    }).thenAccept(
        o -> logger.info(() -> "Печать заказа " + o + " нитью: "
            + Thread.currentThread().getName()));

    cfFetchAndPrintOrder.get();
    logger.info("Заказ получен и напечатан \n");
}
```



Проверьте также методы `acceptEither()` и `acceptEitherAsync()`.

Прикрепление функции обратного вызова, которая выполняется после асинхронной операции и возвращает *void*

Задача пользователя: доставить заказ и уведомить клиента.

Клиент должен быть уведомлен после доставки заказа. Это просто SMS-сообщение с текстом типа: "Уважаемый клиент, ваш заказ доставлен сегодня", поэтому операция уведомления не должна ничего знать о заказе. Такого рода операции могут выполняться методом `thenRun()`, который берет `Runnable` и возвращает `CompletableFuture<Void>`. Давайте посмотрим, как это работает:

```
public static void deliverOrderNotifyCustomer() {
    CompletableFuture<Void> cfDeliverOrder
        = CompletableFuture.runAsync(() -> {

        logger.info(() -> "Заказ был доставлен: "
            + Thread.currentThread().getName());
        Thread.sleep(500);
    });

    CompletableFuture<Void> cfNotifyCustomer
        = cfDeliverOrder.thenRun(() -> logger.info(
            () -> "Уважаемый клиент, ваш заказ доставлен сегодня:"
            + Thread.currentThread().getName()));

    cfNotifyCustomer.get();
    logger.info(() -> "Заказ был доставлен
        и клиент уведомлен \n");
}
```



В целях дальнейшей параллелизации методы `thenApply()`, `thenAccept()` и `thenRun()` сопровождаются методами `thenApplyAsync()`, `thenAcceptAsync()` и `thenRunAsync()`. Каждый из них может опираться на глобальный пул `ForkJoinPool.commonPool()` или пул специализированных нитей исполнения (`Executor`). В то время как `thenApply/Accept/Run()` исполняются в той же нити, что и исполненная ранее операция `CompletableFuture` (или в главной нити), методы `thenApplyAsync/AcceptAsync/RunAsync()` могут исполняться в другой нити (из глобального пула `ForkJoinPool.commonPool()` или пула специализированных нитей (`Executor`)).

Обработка исключений асинхронной операции посредством `exceptionally()`

Задача пользователя: вычислить итоговую сумму заказа. Если что-то пойдет не так, то выбросить исключение `IllegalStateException`.

Снимки экрана на рис. 11.3 иллюстрируют то, как исключения продвигаются в асинхронном конвейере; код в прямоугольниках не выполняется, когда исключение возникает в точке.

<pre>CompletableFuture.supplyAsync(() -> { // Code prone to exception return "result1"; }).thenApply(r1 -> { // Code prone to exception return "result2"; }).thenApply(r2 -> { // Code prone to exception return "result3"; }).thenAccept(r3 -> { // Code prone to exception });</pre> <pre>return "result1"; }).thenApply(r1 -> { // Code prone to exception return "result2"; }).thenApply(r2 -> { // Code prone to exception return "result3"; }).thenAccept(r3 -> { // Code prone to exception });</pre> Исключение в методе <code>supplyAsync()</code>	<pre>CompletableFuture.supplyAsync(() -> { // Code prone to exception return "result1"; }).thenApply(r1 -> { // Code prone to exception return "result2"; }).thenApply(r2 -> { // Code prone to exception return "result3"; }).thenAccept(r3 -> { // Code prone to exception });</pre> <pre>return "result2"; }).thenApply(r2 -> { // Code prone to exception return "result3"; }).thenAccept(r3 -> { // Code prone to exception });</pre> Исключение в первом методе <code>thenApply()</code>

Рис. 11.3

Снимок экрана на рис. 11.4 показывает исключения в методах `thenApply()` и `thenAccept()`.

<pre>CompletableFuture.supplyAsync(() -> { // Code prone to exception return "result1"; }).thenApply(r1 -> { // Code prone to exception return "result2"; }).thenApply(r2 -> { // Code prone to exception return "result3"; }).thenAccept(r3 -> { // Code prone to exception });</pre> <pre>return "result3"; }).thenAccept(r3 -> { // Code prone to exception });</pre> Исключение во втором методе <code>thenApply()</code>	<pre>CompletableFuture.supplyAsync(() -> { // Code prone to exception return "result1"; }).thenApply(r1 -> { // Code prone to exception return "result2"; }).thenApply(r2 -> { // Code prone to exception return "result3"; }).thenAccept(r3 -> { // Code prone to exception });</pre> <pre>});</pre> Исключение в методе <code>thenAccept()</code>

Рис. 11.4

Так, если в методе `supplyAsync()` возникает исключение, то ни один из последующих обратных вызовов не будет вызван. Более того, объект `Future` будет решаться с этим исключением. То же правило применяется для каждого обратного вызова. Если исключение возникает в первом методе `thenApply()`, то последующие методы `thenApply()` и `thenAccept()` вызваны не будут.

Если наша попытка вычислить итоговую сумму заказа заканчивается исключением `IllegalStateException`, то мы можем опереться на обратный вызов `exceptionally()`,

который дает нам шанс на восстановление. Этот метод берет `Function<Throwable,?extends T>` и возвращает `CompletionStage<T>` и, следовательно, `CompletableFuture`. Давайте посмотрим, как это работает:

```
public static void fetchOrderTotalException() {
    CompletableFuture<Integer> cfTotalOrder
        = CompletableFuture.supplyAsync(() -> {

        logger.info(() -> "Вычислить итог: " + Thread.currentThread().getName());

        int surrogate = new Random().nextInt(1000);
        if (surrogate < 500) {
            throw new IllegalStateException(
                "Счет-фактурная служба не откликается");
        }

        return 1000;
    }).exceptionally(ex -> {
        logger.severe(() -> "Исключение: " + ex
            + " Нить: " + Thread.currentThread().getName());
        return 0;
    });

    int result = cfTotalOrder.get();
    logger.info(() -> "Итого: " + result + "\n");
}
```

В случае исключения результат будет выглядеть следующим образом:

```
Вычислить итог: ForkJoinPool.commonPool-worker-3
Исключение: java.lang.IllegalStateException: Счет-фактурная служба
            не откликается Нить: ForkJoinPool.commonPool-worker-3
Итого: 0
```

Давайте рассмотрим еще одну задачу.

Задача пользователя: получить счет, вычислить итоговую сумму и подписать. Если что-то пойдет не так, то выбросить исключение `IllegalStateException` и остановить процесс.

Если мы получаем счет-фактуру с помощью вызова метода `supplyAsync()`, вычисляем итоговую сумму посредством вызова `thenApply()` и подписываем с помощью еще одного вызова `thenApply()`, то мы можем подумать, что правильная имплементация выглядит так:

```
public static void fetchInvoiceTotalSignChainOfException()
    throws InterruptedException, ExecutionException {

    CompletableFuture<String> cfFetchInvoice
        = CompletableFuture.supplyAsync(() -> {
```



```
logger.info(() -> "Получить счет-фактуру: "
    + Thread.currentThread().getName());

int surrogate = new Random().nextInt(1000);
if (surrogate < 500) {
    throw new IllegalStateException(
        "Счет-фактурная служба не откликается");
}

return "Счет-фактура #3344";
}).exceptionally(ex -> {
    logger.severe(() -> "Исключение: " + ex
        + " Нить: " + Thread.currentThread().getName());

    return "[Invoice-Exception]";
}).thenApply(o -> {
    logger.info(() -> "Вычислить итог: "
        + Thread.currentThread().getName());

    int surrogate = new Random().nextInt(1000);
    if (surrogate < 500) {
        throw new IllegalStateException(
            "Подытоживающая служба не откликается");
    }

    return o + " Total: $145";
}).exceptionally(ex -> {
    logger.severe(() -> "Исключение: " + ex
        + " Нить: " + Thread.currentThread().getName());

    return "[Total-Exception]";
}).thenApply(o -> {
    logger.info(() -> "Подписать счет-фактуру: "
        + Thread.currentThread().getName());

    int surrogate = new Random().nextInt(1000);
    if (surrogate < 500) {
        throw new IllegalStateException(
            "Подписывающая служба не откликается");
    }

    return o + " Подписано";
}).exceptionally(ex -> {
    logger.severe(() -> "Исключение: " + ex
        + " Нить: " + Thread.currentThread().getName());
```

```

    return "[Sign-Exception]";
});

String result = cfFetchInvoice.get();
logger.info(() -> "Результат: " + result + "\n");
}

```

Что ж, проблема здесь в том, что мы можем столкнуться вот с таким результатом:

```

[INFO] Получить счет-фактуру: ForkJoinPool.commonPool-worker-3
[SEVERE] Исключение: java.lang.IllegalStateException: Счет-фактурная служба
        не откликается Нить: ForkJoinPool.commonPool-worker-3
[INFO] Вычислить итог: ForkJoinPool.commonPool-worker-3
[INFO] Подписать счет-фактуру: ForkJoinPool.commonPool-worker-3
[SEVERE] Исключение: java.lang.IllegalStateException: Подписывающая служба
        не откликается Нить: ForkJoinPool.commonPool-worker-3
[INFO] Результат: [Sign-Exception]

```

Даже если счет-фактуру получить не получается, мы продолжим подсчитывать итоговую сумму и подпишем ее. Очевидно, что это не имеет смысла. Если счет-фактуру получить не удастся, или итоговую сумму вычислить не получается, то ожидается, что мы прервем процесс. Хотя эта имплементация, возможно, будет уместной, когда нам нужна возможность восстановиться и продолжить, она определенно не подходит для нашего сценария. Нашему сценарию необходима следующая имплементация:

```

public static void fetchInvoiceTotalSignException()
    throws InterruptedException, ExecutionException {

    CompletableFuture<String> cfFetchInvoice
        = CompletableFuture.supplyAsync(() -> {

        logger.info(() -> "Получить счет-фактуру: "
            + Thread.currentThread().getName());

        int surrogate = new Random().nextInt(1000);
        if (surrogate < 500) {
            throw new IllegalStateException(
                "Счет-фактурная служба не откликается");
        }

        return "Счет-фактура #3344";
    }).thenApply(o -> {
        logger.info(() -> "Вычислить итог: " + Thread.currentThread().getName());

        int surrogate = new Random().nextInt(1000);
        if (surrogate < 500) {

```

```
        throw new IllegalStateException(
            "Подытоживающая служба не откликается");
    }

    return o + " Total: $145";
}).thenApply(o -> {
    logger.info(() -> "Подписать счет-фактуру: "
        + Thread.currentThread().getName());

    int surrogate = new Random().nextInt(1000);
    if (surrogate < 500) {
        throw new IllegalStateException(
            "Подписывающая служба не откликается");
    }

    return o + " Подписано";
}).exceptionally(ex -> {
    logger.severe(() -> "Исключение: " + ex
        + " Нить: " + Thread.currentThread().getName());

    return "[No-Invoice-Exception]";
});

String result = cfFetchInvoice.get();
logger.info(() -> "Результат: " + result + "\n");
}
```

На этот раз исключение, возникающее в любом из подразумеваемых `CompletableFuture`, остановит процесс. Вот возможный результат:

```
[INFO ] Получить счет-фактуру: ForkJoinPool.commonPool-worker-3
[SEVERE] Исключение: java.lang.IllegalStateException: Счет-фактурная служба
        не откликается Нить: ForkJoinPool.commonPool-worker-3
[INFO ] Результат: [No-Invoice-Exception]
```

Начиная с JDK 12, случаи исключения могут быть еще больше параллелизованы посредством метода `exceptionallyAsync()`, который может использовать ту же нить исполнения, что и код, ставший причиной исключения, или нить из заданного пула нитей исполнения (`Executor`). Вот пример:

```
public static void fetchOrderTotalExceptionAsync() {
    ExecutorService executor = Executors.newSingleThreadExecutor();

    CompletableFuture<Integer> totalOrder
        = CompletableFuture.supplyAsync(() -> {

        logger.info(() -> "Вычислить итог: " + Thread.currentThread().getName());
```

```

int surrogate = new Random().nextInt(1000);
if (surrogate < 500) {
    throw new IllegalStateException("Вычислительная служба не откликается");
}

return 1000;
}).exceptionallyAsync(ex -> {
    logger.severe(() -> "Исключение: " + ex
        + " Нить: " + Thread.currentThread().getName());

    return 0;
}, executor);

int result = totalOrder.get();
logger.info(() -> "Итого: " + result + "\n");
executor.shutdownNow();
}

```

Результат показывает, что код, ставший причиной исключения, был исполнен нитью с именем `ForkJoinPool.commonPool-worker-3`, тогда как код исключения был исполнен нитью из заданного пула нитей исполнения с именем `pool-1-thread-1`:

Вычислить итог: `ForkJoinPool.commonPool-worker-3`

Исключение: `java.lang.IllegalStateException: Вычислительная служба не откликается` Нить: `pool-1-thread-1`

Итого: 0

Метод `exceptionallyCompose()` JDK 12

Задача пользователя: получить IP-адрес принтера посредством службы печати или откатить к IP-адресу резервного принтера. Другими словами, когда эта стадия завершается с исключением, она должна быть выстроена с использованием результатов поставляемой функции, примененной к исключению этой стадии.

У нас есть объект `CompletableFuture`, который извлекает IP-адрес принтера, управляемого службой печати. Если эта служба не откликается, то он выбрасывает исключение следующим образом:

```

CompletableFuture<String> cfServicePrinterIp
    = CompletableFuture.supplyAsync(() -> {

        int surrogate = new Random().nextInt(1000);
        if (surrogate < 500) {
            throw new IllegalStateException("Служба печати не откликается");
        }

        return "192.168.1.0";
    });

```

У нас также есть объект `CompletableFuture`, который извлекает IP-адрес резервного принтера:

```
CompletableFuture<String> cfBackupPrinterIp
    = CompletableFuture.supplyAsync(() -> {

        return "192.192.192.192";

    });
```

Теперь, если служба печати недоступна, мы должны опереться на резервный принтер. Это можно выполнить посредством метода `exceptionallyCompose()` JDK 12 следующим образом:

```
CompletableFuture<Void> printInvoice
    = cfServicePrinterIp.exceptionallyCompose(th -> {

        logger.severe(() -> "Исключение: " + th
            + " Нить: " + Thread.currentThread().getName());

        return cfBackupPrinterIp;

    }).thenAccept((ip) -> logger.info(() -> "Печать по адресу: " + ip));
```

Вызов метода `printInvoice.get()` может привести к одному из следующих ниже результатов:

◆ если служба печати доступна:

```
[INFO] Печать по адресу: 192.168.1.0
```

◆ если служба печати недоступна:

```
[SEVERE] Исключение: java.util.concurrent.CompletionException ...
[INFO] Печать по адресу: 192.192.192.192
```

Для дальнейшей параллелизации мы можем опереться на метод `exceptionallyComposeAsync()`.

Обработка исключений асинхронной операции посредством метода *handle()*

Задача пользователя: вычислить итоговую сумму заказа. Если что-то пойдет не так, то выбросить исключение `IllegalStateException`.

Иногда нужно исполнить блок кода исключения, даже если исключение не наступило подобно условию `finally` блока `try-catch`. Это возможно с помощью метода обратного вызова `handle()`. Данный метод вызывается независимо от того, произошло ли исключение, и похож на связку `catch + finally`. Он берет бифункцию `BiFunction<? super T, Throwable, ? extends U>`, используемую для вычисления значения возвращаемого этапа завершения `CompletionStage`, и возвращает `CompletionStage<U>` (`U` — это тип значения, возвращаемого функцией).

Давайте посмотрим на то, как это работает:

```
public static void fetchOrderTotalHandle() {
    CompletableFuture<Integer> totalOrder
        = CompletableFuture.supplyAsync(() -> {

        logger.info(() -> "Вычислить итог: "
            + Thread.currentThread().getName());

        int surrogate = new Random().nextInt(1000);
        if (surrogate < 500) {
            throw new IllegalStateException("Вычислительная служба не откликается");
        }

        return 1000;
    }).handle((res, ex) -> {
        if (ex != null) {
            logger.severe(() -> "Исключение: " + ex
                + " Нить: " + Thread.currentThread().getName());

            return 0;
        }

        if (res != null) {
            int vat = res * 24 / 100;
            res += vat;
        }

        return res;
    });

    int result = totalOrder.get();
    logger.info(() -> "Итого: " + result + "\n");
}
```

Обратите внимание, что `res` будет равно `null`; в противном случае `ex` будет равно `null`, если произойдет исключение.

Если нам нужно завершить обработку исключения, мы можем продолжить посредством метода `completeExceptionally()`, как в следующем примере:

```
CompletableFuture<Integer> cf = new CompletableFuture<>();
...
cf.completeExceptionally(new RuntimeException("Ops!"));
...
cf.get(); // ExecutionException : RuntimeException
```

Отменить исполнение и выбросить исключение `CancellationException` можно посредством метода `cancel()`:

```
CompletableFuture<Integer> cf = new CompletableFuture<>();
...

// неважно, установлен ли аргумент равным true либо false
cf.cancel(true/false);
...
cf.get(); // исключение CancellationException
```

Явное завершение экземпляра класса *CompletableFuture*

Экземпляр класса `CompletableFuture` может быть явно завершен с помощью методов `complete(T value)`, `completeAsync(Supplier<? extends T> supplier)` и `completeAsync(Supplier<? extends T> supplier, Executor executor)`. `T` — это значение, возвращаемое методом `get()`. Здесь это метод, который создает экземпляр класса `CompletableFuture` и возвращает его немедленно. Еще одна нить исполнения отвечает за исполнение некоторых налоговых вычислений и завершение экземпляра класса `CompletableFuture` с соответствующим результатом:

```
public static CompletableFuture<Integer> taxes() {
    CompletableFuture<Integer> completableFuture
        = new CompletableFuture<>();

    new Thread(() -> {
        int result = new Random().nextInt(100);
        Thread.sleep(10);

        completableFuture.complete(result);
    }).start();

    return completableFuture;
}
```

И давайте вызовем этот метод:

```
logger.info("Вычисление налогов...");

CompletableFuture<Integer> cfTaxes = CustomerAsyncns.taxes();

while (!cfTaxes.isDone()) {
    logger.info("По-прежнему вычисляются...");
}

int result = cfTaxes.get();
logger.info(() -> "Результат: " + result);
```

Возможный результат будет таким:

```
[14:09:40] [INFO ] Computing taxes ...
[14:09:40] [INFO ] Still computing ...
[14:09:40] [INFO ] Still computing ...
...
[14:09:40] [INFO ] Still computing ...
[14:09:40] [INFO ] Result: 17
```

Если мы уже знаем результат `CompletableFuture`, то можем вызвать метод `completedFuture(U value)`, как в следующем примере:

```
CompletableFuture<String> completableFuture
    = CompletableFuture.completedFuture("Как дела?");
```

```
String result = completableFuture.get();
logger.info(() -> "Результат: " + result); // Результат: Как дела?
```



Кроме того, проверьте документацию по методам `whenComplete()` и `whenCompleteAsync()`.

217. Сочетание нескольких экземпляров класса *CompletableFuture*

В большинстве случаев сочетать экземпляры класса `CompletableFuture` можно с помощью следующих методов:

- ◆ `thenCompose()`;
- ◆ `thenCombine()`;
- ◆ `allOf()`;
- ◆ `anyOf()`.

Комбинируя экземпляры класса `CompletableFuture`, мы можем формировать сложные асинхронные решения. Благодаря этому несколько экземпляров класса `CompletableFuture` могут сочетать свои полномочия для достижения общей цели.

Сочетание посредством метода *thenCompose()*

Допустим, что у нас есть следующие два экземпляра класса `CompletableFuture` во вспомогательном классе с именем `CustomerAsyncs`:

```
private static CompletableFuture<String> fetchOrder(String customerId) {
    return CompletableFuture.supplyAsync(() -> {
        return "Заказ клиента " + customerId;
    });
}
```



```
private static CompletableFuture<Integer> computeTotal(String order) {  
    return CompletableFuture.supplyAsync(() -> {  
        return order.length() + new Random().nextInt(1000);  
    });  
}
```

Теперь мы хотим получить заказ от некоторого клиента и, как только заказ будет доступен, вычислить итоговую сумму этого заказа. Это означает, что нам нужно вызвать метод `fetchOrder()` и затем метод `computeTotal()`. Мы можем сделать это посредством метода `thenApply()`:

```
CompletableFuture<CompletableFuture<Integer>> cfTotal  
    = fetchOrder(customerId).thenApply(o -> computeTotal(o));
```

```
int total = cfTotal.get().get();
```

Очевидно, что это решение не очень удобное, т. к. результат имеет тип `CompletableFuture<CompletableFuture<Integer>>`. Во избежание вложенности экземпляров класса `CompletableFuture` мы можем опереться на метод `thenCompose()` следующим образом:

```
CompletableFuture<Integer> cfTotal  
    = fetchOrder(customerId).thenCompose(o -> computeTotal(o));
```

```
int total = cfTotal.get();
```

```
// например, Итого: 734  
logger.info(() -> "Итого: " + total);
```



Всякий раз, когда нам нужно получить сглаженный результат из цепочки экземпляров `CompletableFuture`, мы можем использовать метод `thenCompose()`. Благодаря этому мы избегаем вложенных экземпляров класса `CompletableFuture`.

Дальнейшую параллелизацию можно получить с помощью метода `thenComposeAsync()`.

Сочетание посредством метода *thenCombine()*

В то время как метод `thenCompose()` полезен для выстраивания в цепочку двух зависимых экземпляров класса `CompletableFuture`, метод `thenCombine()` полезен для цепочки двух независимых экземпляров класса `CompletableFuture`. Когда оба экземпляра класса `CompletableFuture` завершены, мы можем продолжить.

Допустим, что у нас есть следующие два экземпляра класса `CompletableFuture`:

```
private static CompletableFuture<Integer> computeTotal(String order) {  
    return CompletableFuture.supplyAsync(() -> {  
        return order.length() + new Random().nextInt(1000);  
    });  
}
```

```
private static CompletableFuture<String> packProducts(String order) {
    return CompletableFuture.supplyAsync(() -> {
        return "Заказ: " + order
            + " | Продукт 1, Продукт 2, Продукт 3, ... ";
    });
}
```

Для того чтобы доставить заказ клиенту, нам нужно вычислить итоговую сумму (для выдачи счета-фактуры) и упаковать заказанные продукты. Оба действия могут выполняться параллельно. В конце мы доставляем посылку, содержащую заказанные продукты и счет-фактуру. Достичь этого посредством метода `thenCombine()` можно следующим образом:

```
CompletableFuture<String> cfParcel = computeTotal(order)
    .thenCombine(packProducts(order), (total, products) -> {
        return "Посылка-[ " + products + " Счет-фактура: $" + total + " ]";
    });

String parcel = cfParcel.get();

// например, Доставка: Посылка-[Заказ: #332 | Продукт 1, Продукт 2,
// Продукт 3, ... Счет-фактура: $314]
logger.info(() -> "Доставка: " + parcel);
```

Функция обратного вызова, передаваемая в метод `thenCombine()`, будет вызвана после завершения обоих экземпляров класса `CompletableFuture`.

Если нам нужно лишь сделать что-то, когда два экземпляра класса `CompletableFuture` завершаются нормально (и тот и другой этапы), то мы можем опереться на метод `thenAcceptBoth()`. Данный метод возвращает новый экземпляр класса `CompletableFuture`, который исполняется, причем два результата передаются в качестве аргументов для предоставленного действия. Два результата — это и тот и другой заданные этапы (они должны завершиться нормально). Вот пример:

```
ompletableFuture<Void> voidResult = CompletableFuture
    .supplyAsync(() -> "Подбери")
    .thenAcceptBoth(CompletableFuture.supplyAsync(() -> "меня"),
        (pick, me) -> System.out.println(pick + me));
```

Если результаты этих двух экземпляров класса `CompletableFuture` не нужны, то метод `runAfterBoth()` будет гораздо предпочтительнее.

Сочетание посредством метода `allOf()`

Допустим, что мы хотим скачать следующий список счетов-фактур:

```
List<String> invoices = Arrays.asList("#2334", "#122", "#55");
```

Поставленную задачу можно рассматривать как связку независимых задач, способных выполняться в параллельном режиме, поэтому мы можем сделать это с помощью экземпляра класса `CompletableFuture` следующим образом:

```
public static CompletableFuture<String> downloadInvoices(String invoice) {
    return CompletableFuture.supplyAsync(() -> {
        logger.info(() -> "Скачивание счет-фактуры: " + invoice);

        return "Скачана счет-фактура: " + invoice;
    });
}
```

```
CompletableFuture<String> [] cfInvoices = invoices.stream()
    .map(CustomerAsyncs::downloadInvoices)
    .toArray(CompletableFuture[]::new);
```

На этом месте мы имеем массив экземпляров класса `CompletableFuture` и, следовательно, массив асинхронных вычислений. Далее мы хотим, чтобы все они работали параллельно. Этого можно достичь с помощью метода `allOf(CompletableFuture<?>... cfs)`. Результат состоит из `CompletableFuture<Void>` и получается следующим образом:

```
CompletableFuture<Void> cfDownloaded = CompletableFuture.allOf(cfInvoices);
cfDownloaded.get();
```

Очевидно, что результат метода `allOf()` не очень полезен. Что можно сделать с `CompletableFuture<Void>`? Безусловно, существует много задач, когда нам нужны результаты каждого вычисления, участвующего в этой параллелизации, и поэтому мы должны получить решение для извлечения результатов вместо того, чтобы опираться на `CompletableFuture<Void>`.

Мы можем решить эту задачу посредством метода `thenApply()` следующим образом:

```
List<String> results = cfDownloaded.thenApply(e -> {
    List<String> downloaded = new ArrayList<>();

    for (CompletableFuture<String> cfInvoice: cfInvoices) {
        downloaded.add(cfInvoice.join());
    }

    return downloaded;
}).get();
```



Метод `join()` похож на метод `get()`, но если опорный объект `CompletableFuture` завершается с исключением, то он создает непроверенное исключение.

Поскольку мы вызываем `join()` после того, как все задействованные операции `CompletableFuture` завершены, блокирующая точка отсутствует.

Возвращаемый список `List<String>` содержит результаты, полученные при вызове метода `downloadInvoices()` следующим образом:

Скачана счет-фактура: #2334

Скачана счет-фактура: #122

Скачана счет-фактура: #55

Сочетание посредством метода *anyOf()*

Допустим, что мы хотим организовать розыгрыш для наших клиентов:

```
List<String> customers = Arrays.asList(
    "#1", "#4", "#2", "#7", "#6", "#5"
);
```

Мы можем начать решать эту задачу, определив следующий тривиальный метод:

```
public static CompletableFuture<String> raffle(String customerId) {
    return CompletableFuture.supplyAsync(() -> {
        Thread.sleep(new Random().nextInt(5000));

        return customerId;
    });
}
```

Теперь мы можем создать массив экземпляров `CompletableFuture<String>`:

```
CompletableFuture<String>[] cfCustomers = customers.stream()
    .map(CustomerAsyncs::raffle)
    .toArray(CompletableFuture[]::new);
```

Для того чтобы найти победителя розыгрыша, мы хотим выполнять `cfCustomers` параллельно, и первый завершившийся экземпляр `CompletableFuture` будет победителем. Поскольку метод `raffle()` блокирует в течение случайного числа секунд, победитель будет выбран случайным образом. Нас не интересуют остальные экземпляры класса `CompletableFuture`, поэтому они должны быть завершены сразу же после того, как был выбран победитель.

Это работа как раз для метода `anyOf(CompletableFuture<?>... cfs)`. Он возвращает новый экземпляр класса `CompletableFuture`, который завершен, когда любой из вовлеченных экземпляров класса `CompletableFuture` завершается. Давайте посмотрим, как это работает:

```
CompletableFuture<Object> cfWinner = CompletableFuture.anyOf(cfCustomers);
```

```
Object winner = cfWinner.get();
```

```
// например, Победитель: #2
logger.info(() -> "Победитель: " + winner);
```



Обратите внимание на сценарии, опирающиеся на `CompletableFuture`, которые возвращают результаты разных типов. Поскольку метод `anyOf()` возвращает `CompletableFuture<Object>`, выяснить, какого типа экземпляры `CompletableFuture` завершились первыми, очень трудно.

218. Оптимизирование занятого ожидания

Технический прием *занятого ожидания* (также известный как цикл занятости или спин-ожидание) состоит из цикла, который проверяет условие (в типичной ситуации, флаговое условие). Например, следующий ниже цикл ожидает, когда служба запустится:

```
private volatile boolean serviceAvailable;
...
while (!serviceAvailable) {}
```

Среда Java 9 ввела метод `Thread.onSpinWait()`. Он представляет собой горячую точку, которая намекает JVM о том, что следующий код находится в цикле занятости:

```
while (!serviceAvailable) {
    Thread.onSpinWait();
}
```



Инструкция Intel SSE2 PAUSE предусмотрена именно по этой причине. Дополнительные сведения см. в официальной документации Intel. Также обратитесь к материалу по ссылке: <https://software.intel.com/en-us/articles/benefitting-power-and-performance-sleep-loops>.

Если мы добавим этот цикл `while` в контекст, то получим следующий класс:

```
public class StartService implements Runnable {
    private volatile boolean serviceAvailable;

    @Override
    public void run() {
        System.out.println("Подождите, пока служба будет доступна...");

        while (!serviceAvailable) {
            // Использовать намек спин-ожидания
            // (попросить процессор оптимизировать ресурс).
            // Это должно работать лучше, если опорное
            // аппаратное обеспечение поддерживает указанный намек
            Thread.onSpinWait();
        }

        serviceRun();
    }

    public void serviceRun() {
        System.out.println("Служба работает...");
    }
}
```

```

public void setServiceAvailable(boolean serviceAvailable) {
    this.serviceAvailable = serviceAvailable;
}
}

```

И мы можем легко проверить его (не ожидайте, что увидите эффект от метода `onSpinWait()`):

```

StartService startService = new StartService();
new Thread(startService).start();

```

```

Thread.sleep(5000);

```

```

startService.setServiceAvailable(true);

```

219. Отмена операции

Отмена операции — это часто встречающийся технический прием, используемый для принудительной остановки или завершения операции, которая выполняется в данное время. Отмененная операция не будет завершена естественным образом. Отмена не должна иметь никакого влияния на уже выполненную операцию. Думайте об этом как о кнопке **Отмена** в графическом интерфейсе пользователя.

Java не предоставляет упреждающего способа остановки нити исполнения. Поэтому для отмены операции обычно опираются на цикл, который использует флаговое условие. Ответственность операции лежит в периодической проверке этого флага, и когда она обнаруживает, что указанный флаг установлен, она должна остановиться как можно быстрее. Следующий фрагмент кода является примером этого:

```

public class RandomList implements Runnable {
    private volatile boolean cancelled;
    private final List<Integer> randomness = new CopyOnWriteArrayList<>();
    private final Random rnd = new Random();

    @Override
    public void run() {
        while (!cancelled) {
            randomness.add(rnd.nextInt(100));
        }
    }

    public void cancel() {
        cancelled = true;
    }

    public List<Integer> getRandoms() {
        return randomness;
    }
}

```

Здесь в центре внимания находится переменная `canceled`. Обратите внимание, что эта переменная была объявлена как `volatile` (это так называемый более легковесный механизм синхронизации). Будучи волатильной переменной, она не кэшируется нитями исполнения, и операции на ней не переупорядочиваются в памяти; поэтому нить не может видеть старое значение. Любая нить исполнения, читающая волатильное поле, будет видеть самое последнее записанное значение. Это именно то, что нам нужно, чтобы сообщить об отменяющем действии всем работающим нитям исполнения, которые заинтересованы в этом действии. Схема на рис. 11.5 показывает принцип работы волатильных и неволатильных переменных.

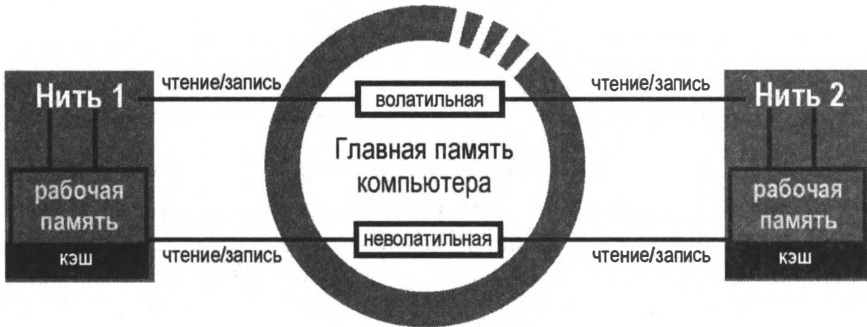


Рис. 11 5

Обратите внимание, что волатильные переменные не подходят для сценариев чтения-модификации-записи. Для таких сценариев мы будем опираться на атомарные переменные (например, `AtomicBoolean`, `AtomicInteger`, `AtomicReference` и т. д.).

Теперь давайте предоставим простой фрагмент кода для отмены операции, имплементированной в классе `RandomList`:

```
RandomList rl = new RandomList();

ExecutorService executor = Executors.newFixedThreadPool(10);

for (int i = 0; i < 100; i++) {
    executor.execute(rl);
}

Thread.sleep(100);

rl.cancel();

System.out.println(rl.getRandoms());
```

220. Переменные типа *ThreadLocal*

Нити исполнения среды Java используют одну и ту же память совместно, но иногда нам требуется память, специально выделенная для каждой нити. Java предоставляет класс `ThreadLocal` в качестве подхода для хранения и извлечения значений для каж-

дой нити исполнения в отдельности. Один экземпляр класса `ThreadLocal` может хранить и извлекать значения нескольких нитей исполнения. Если нить `A` хранит значение `X` и нить `B` хранит значение `Y` в одном экземпляре класса `ThreadLocal`, то позже нить `A` получает значение `X`, а нить `B` — значение `Y`.

Класс `ThreadLocal` Java обычно используется в следующих двух сценариях:

- ◆ для предоставления экземпляров в расчете на нить исполнения (нитебезопасность и эффективность памяти);
- ◆ для предоставления контекста в расчете на нить исполнения.

Давайте посмотрим на задачи для каждого сценария в следующих разделах.

Экземпляры в расчете на нить исполнения

Допустим, что у нас есть однопоточное приложение, которое использует глобальную переменную типа `StringBuilder`. Для того чтобы преобразовать приложение в многопоточное, мы должны справиться с типом `StringBuilder`, который не является нитебезопасным.

В принципе, у нас есть несколько подходов, таких как синхронизация и класс `StringBuffer` или другие подходы. Однако мы можем использовать и класс `ThreadLocal`. Главная идея здесь заключается в предоставлении отдельного экземпляра `StringBuilder` для каждой нити исполнения. Используя класс `ThreadLocal`, мы можем сделать это следующим образом:

```
private static final ThreadLocal<StringBuilder>
    threadLocal = new ThreadLocal<>() {

    @Override
    protected StringBuilder initialValue() {
        return new StringBuilder("ThreadSafe ");
    }
};
```

Начальное значение текущей нити исполнения для этой локальной для нити переменной устанавливается посредством метода `initialValue()`. В Java 8 это можно переписать с помощью метода `withInitial()` следующим образом:

```
private static final ThreadLocal<StringBuilder> threadLocal
    = ThreadLocal.<StringBuilder> withInitial(() -> {

    return new StringBuilder("Thread-safe ");
});
```

Работа с классом `ThreadLocal` выполняется с помощью методов `get()` и `set()`. Каждый вызов метода `set()` сохраняет данное значение в области памяти, к которой имеет доступ только текущая нить исполнения. Позже вызов метода `get()` будет возвращать значение из этой области. В дополнение к этому, после выполнения

работы рекомендуется избегать утечек памяти, вызывая метод `remove()` или метод `set(null)` в экземпляре класса `ThreadLocal`.

Давайте посмотрим на то, как работает `ThreadLocal` с использованием `Runnable`:

```
public class ThreadSafeStringBuilder implements Runnable {
    private static final Logger logger =
        Logger.getLogger(ThreadSafeStringBuilder.class.getName());
    private static final Random rnd = new Random();

    private static final ThreadLocal<StringBuilder> threadLocal
        = ThreadLocal.<StringBuilder> withInitial(() -> {

        return new StringBuilder("Thread-safe ");
    });

    @Override
    public void run() {
        logger.info(() -> "-> " + Thread.currentThread().getName()
            + " [" + threadLocal.get() + "]");

        Thread.sleep(rnd.nextInt(2000));

        // threadLocal.set(new StringBuilder(
        // Thread.currentThread().getName()));
        threadLocal.get().append(Thread.currentThread().getName());

        logger.info(() -> "-> " + Thread.currentThread().getName()
            + " [" + threadLocal.get() + "]");
        threadLocal.set(null);
        // threadLocal.remove();
        logger.info(() -> "-> " + Thread.currentThread().getName()
            + " [" + threadLocal.get() + "]");
    }
}
```

И давайте протестируем его с помощью нескольких нитей исполнения:

```
ThreadSafeStringBuilder threadSafe = new ThreadSafeStringBuilder();
```

```
for (int i = 0; i < 3; i++) {
    new Thread(threadSafe, "thread-" + i).start();
}
```

Результат показывает, что каждая нить обращается к своему собственному экземпляру `StringBuilder`:

```
[14:26:39] [INFO] -> thread-1 [Thread-safe ]
[14:26:39] [INFO] -> thread-0 [Thread-safe ]
[14:26:39] [INFO] -> thread-2 [Thread-safe ]
```

```
[14:26:40] [INFO] -> thread-0 [Thread-safe thread-0]
[14:26:40] [INFO] -> thread-0 [null]
[14:26:41] [INFO] -> thread-1 [Thread-safe thread-1]
[14:26:41] [INFO] -> thread-1 [null]
[14:26:41] [INFO] -> thread-2 [Thread-safe thread-2]
[14:26:41] [INFO] -> thread-2 [null]
```



В сценариях, подобных предыдущему, можно также использовать класс `ExecutorService`.

Вот еще один фрагмент кода, который предоставляет каждой нити исполнения подключение JDBC:

```
private static final ThreadLocal<Connection> connections
    = ThreadLocal.<Connection> withInitial(() -> {

    try {
        return DriverManager.getConnection("jdbc:mysql://...");
    } catch (SQLException ex) {
        throw new RuntimeException("Приобрести подключение не получилось!", ex);
    }
});

public static Connection getConnection() {
    return connections.get();
}
```

Контекст в расчете на нить исполнения

Допустим, что мы имеем следующий ниже класс `Order`:

```
public class Order {
    private final int customerId;

    public Order(int customerId) {
        this.customerId = customerId;
    }

    // геттер и toString() опущены для краткости
}
```

И мы пишем класс `CustomerOrder` следующим образом:

```
public class CustomerOrder implements Runnable {
    private static final Logger logger
        = Logger.getLogger(CustomerOrder.class.getName());
    private static final Random rnd = new Random();

    private static final ThreadLocal<Order>
        customerOrder = new ThreadLocal<>();
```

```
private final int customerId;

public CustomerOrder(int customerId) {
    this.customerId = customerId;
}

@Override
public void run() {
    logger.info(() -> "Заданный id клиента: " + customerId
        + " | " + customerOrder.get()
        + " | " + Thread.currentThread().getName());

    customerOrder.set(new Order(customerId));

    try {
        Thread.sleep(rnd.nextInt(2000));
    } catch (InterruptedException ex) {
        Thread.currentThread().interrupt();
        logger.severe(() -> "Исключение: " + ex);
    }

    logger.info(() -> "Заданный id клиента: " + customerId
        + " | " + customerOrder.get()
        + " | " + Thread.currentThread().getName());

    customerOrder.remove();
}
}
```

Для каждого customerId у нас есть выделенная нить исполнения, которой мы управляем:

```
CustomerOrder col = new CustomerOrder(1);
CustomerOrder co2 = new CustomerOrder(2);
CustomerOrder co3 = new CustomerOrder(3);
```

```
new Thread(col).start();
new Thread(co2).start();
new Thread(co3).start();
```

Таким образом, каждая нить исполнения модифицирует некоторый экземпляр CustomerOrder (для каждого экземпляра существует конкретная нить исполнения).

Метод run() извлекает заказ для заданного customerId и сохраняет его в переменной типа ThreadLocal, используя метод set().

Возможный результат будет выглядеть следующим образом:

```
[14:48:20] [INFO]
    Заданный id клиента: 3 | null | Thread-2
```

```
[14:48:20] [INFO]
    Заданный id клиента: 2 | null | Thread-1
[14:48:20] [INFO]
    Заданный id клиента: 1 | null | Thread-0
[14:48:20] [INFO]
    Заданный id клиента: 2 | Order{customerId=2} | Thread-1
[14:48:21] [INFO]
    Заданный id клиента: 3 | Order{customerId=3} | Thread-2
[14:48:21] [INFO]
    Заданный id клиента: 1 | Order{customerId=1} | Thread-0
```

В сценариях, подобных предыдущему, избегайте использования класса `ExecutorService`. Нет никакой гарантии, что каждый объект `Runnable` (заданного `customerId`) будет обрабатываться одной и той же нитью исполнения при каждом исполнении. Это может привести к странным результатам.

221. Атомарные переменные

Наивный подход для подсчета всех чисел от 1 до 1 млн посредством объекта `Runnable` может выглядеть следующим образом:

```
public class Incrementator implements Runnable {
    public [static] int count = 0;

    @Override
    public void run() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

И давайте раскрутим пять нитей исполнения, которые будут увеличивать переменную `count` одновременно:

```
Incrementator nonAtomicInc = new Incrementator();
ExecutorService executor = Executors.newFixedThreadPool(5);

for (int i = 0; i < 1_000_000; i++) {
    executor.execute(nonAtomicInc);
}
```

Но если мы выполним этот код несколько раз, то получим разные результаты, как показано ниже:

```
997776, 997122, 997681 ...
```

Итак, почему же мы не получили ожидаемый результат 1 000 000? Причина в том, что приращение `count++` не является атомарной операцией/действием. Оно состоит из трех атомарных байт-кодовых инструкций:

```
iload_1
iinc 1, 1
istore_1
```

Во время этой операции одна нить исполнения читает значение `count` и увеличивает его на единицу, а другая нить читает более старое значение, что и приводит к неправильному результату. В многонитевом приложении планировщик может останавливать исполнение текущей нити между каждой из этих байт-кодовых инструкций и запускать новую нить, которая работает на той же переменной. Мы можем исправить эту ситуацию с помощью синхронизации или, что еще лучше, посредством атомарных переменных.

Классы атомарных переменных доступны в `java.util.concurrent.atomic`. Они представляют собой классы-обертки, которые ограничивают область соперничества до одной-единственной переменной; они являются намного легковеснее, чем синхронизация Java, и основаны на CAS (Compare and Swap — сравнить и поменять местами: современные процессоры поддерживают это техническое решение, в котором они сравнивают содержимое заданного участка памяти с заданным значением и обновляют его до нового значения, если текущее значение равно ожидаемому значению). Эти классы являются атомарными составными действиями, которые влияют на одно значение в беззамковом стиле, подобно волатильным переменным. Наиболее часто используемыми атомарными переменными являются скаляры:

- ◆ `AtomicInteger;`
- ◆ `AtomicLong;`
- ◆ `AtomicBoolean;`
- ◆ `AtomicReference.`

И, следующие ниже, предназначенные для массивов:

- ◆ `AtomicIntegerArray;`
- ◆ `AtomicLongArray;`
- ◆ `AtomicReferenceArray.`

Давайте перепишем наш пример посредством `AtomicInteger`:

```
public class AtomicIncrementator implements Runnable {
    public static AtomicInteger count = new AtomicInteger();

    @Override
    public void run() {
        count.incrementAndGet();
    }
}
```

```

public int getCount() {
    return count.get();
}

```

Обратите внимание, что вместо `count++` мы написали `count.incrementAndGet()`. Это лишь один из методов, предусмотренных `AtomicInteger`. Указанный метод атомарно наращивает переменную и возвращает новое значение. На этот раз переменная `count` будет равна 1 млн.

В табл. 11.1 приведены несколько часто используемых методов класса `AtomicInteger`. Левый столбец содержит методы, тогда как правый столбец — значение неатомарной операции:

```

AtomicInteger ai = new AtomicInteger(0);    // атомарная
int i = 0;    // неатомарная

// и
int q = 5;
int r;

// и
int e = 0;
boolean b;

```

Таблица 11.1

Атомарные операции	Неатомарные противоположности
<code>r = ai.get();</code>	<code>r = i;</code>
<code>ai.set(q);</code>	<code>i = q;</code>
<code>r = ai.incrementAndGet();</code>	<code>r = ++i;</code>
<code>r = ai.getAndIncrement();</code>	<code>r = i++;</code>
<code>r = ai.decrementAndGet();</code>	<code>r = --i;</code>
<code>r = ai.getAndDecrement();</code>	<code>r = i--;</code>
<code>r = ai.addAndGet(q);</code>	<code>i = i + q; r = i;</code>
<code>r = ai.getAndAdd(q);</code>	<code>r = i; i = i + q;</code>
<code>r = ai.getAndSet(q);</code>	<code>r = i; i = q;</code>
<code>b = ai.compareAndSet(e, q);</code>	<pre> if (i == e) { i = q; return true; } else { return false; } </pre>

Решим несколько задач посредством атомарных операций.

◆ Обновление элементов массива посредством метода

```

updateAndGet(IntUnaryOperator updateFunction):
// [9, 16, 4, 25]

```

```
AtomicIntegerArray atomicArray
    = new AtomicIntegerArray(new int[] {3, 4, 2, 5});

for (int i = 0; i < atomicArray.length(); i++) {
    atomicArray.updateAndGet(i, elem -> elem * elem);
}
```

◆ **Обновление одного целого числа посредством метода**

`updateAndGet(IntUnaryOperator updateFunction):`

```
// 15
AtomicInteger nr = new AtomicInteger(3);
int result = nr.updateAndGet(x -> 5 * x);
```

◆ **Обновление одного целого числа посредством метода** `accumulateAndGet(int x, IntBinaryOperator accumulatorFunction):`

```
// 15
AtomicInteger nr = new AtomicInteger(3);
// x = 3, y = 5
int result = nr.accumulateAndGet(5, (x, y) -> x * y);
```

◆ **Обновление одного целого числа посредством метода** `addAndGet(int delta):`

```
// 7
AtomicInteger nr = new AtomicInteger(3);
int result = nr.addAndGet(4)
```

◆ **Обновление одного целого числа посредством метода**

```
compareAndSet(int expectedValue, int newValue):
// 5, true
AtomicInteger nr = new AtomicInteger(3);
boolean wasSet = nr.compareAndSet(3, 5);
```

Начиная с JDK 9, классы атомарных переменных были обогащены несколькими методами, такими как `get/setPlain()`, `get/setOpaque()`, `getAcquire()` и их компаньонами. Для того чтобы получить представление об этих методах, взгляните на статью "Использование режимов распорядка памяти JDK 9" Дуга Ли (Doug Lea, "Using JDK 9 Memory Order Modes"), доступную по адресу <http://gee.cs.oswego.edu/dl/html/j9mm.html> на момент написания этой книги.

Сумматоры и накопители

Согласно документации по API Java, в случаях многонитевых приложений, которые часто обновляются, но читаются реже, рекомендуется вместо классов `AtomicFoo` опираться на классы `LongAdder`, `DoubleAdder`, `LongAccumulator` и `DoubleAccumulator`. В таких сценариях эти классы предназначены для оптимизации использования нити исполнения.

Это означает, что вместо класса `AtomicInteger` для подсчета целых чисел от 1 до 1 млн мы можем применить класс `LongAdder` следующим образом:

```
public class AtomicAdder implements Runnable {
    public static LongAdder count = new LongAdder();
```

```

@Override
public void run() {
    count.add(1);
}

public long getCount() {
    return count.sum();
}
}

```

В качестве альтернативы мы можем использовать класс `LongAccumulator`:

```

public class AtomicAccumulator implements Runnable {
    public static LongAccumulator count = new LongAccumulator(Long::sum, 0);

    @Override
    public void run() {
        count.accumulate(1);
    }

    public long getCount() {
        return count.get();
    }
}

```

Классы `LongAdder` и `DoubleAdder` подходят для сценариев, предусматривающих добавления (операции, специфичные для сложения значений), в то время как классы `LongAccumulator` и `DoubleAccumulator` подходят для сценариев, которые опираются на заданную функцию для сочетания значений.

222. Класс *ReentrantLock*

Интерфейс `Lock` содержит набор операций с замком, которые явным образом используются для точной настройки процесса приобретения замка (он обеспечивает больший контроль, чем использование внутренних замков). Среди них у нас есть опрашиваемое, безусловное, своевременное и прерываемое приобретение замка. Интерфейс `Lock` выставляет наружу объекты `Future` ключевого слова `synchronized` с дополнительными способностями. Интерфейс `Lock` показан в следующем фрагменте кода:

```

public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long timeout, TimeUnit unit)
        throws InterruptedException;
    void unlock();
    Condition newCondition();
}

```


Одной из имплементаций интерфейса `Lock` является класс `ReentrantLock`. Реентерабельный замок (замок с повторным входом) действует следующим образом: когда нить исполнения входит в замок в первый раз, число удерживаний устанавливается равным единице. Прежде чем снять замок, нить исполнения может повторно войти в замок, приводя к увеличению счетчика удерживаний на единицу для каждого входа. Каждый запрос на снятие замка уменьшает число удерживаний на единицу, и, когда число удерживаний становится равно нулю, запертый ресурс открывается.

Имея те же координаты, что и ключевое слово `synchronized`, класс `ReentrantLock` подчиняется следующей идиоме имплементации:

```
Lock / ReentrantLock lock = new ReentrantLock();
...
lock.lock();

try {
    ...
} finally {
    lock.unlock();
}
```



В случае несправедливых замков порядок, в котором нитям исполнения предоставляется доступ, не определен. Если замок должен быть справедливым (отдать приоритет нити исполнения, которая ждала дольше всего), следует использовать конструктор `ReentrantLock(boolean fair)`.

Просуммировать целые числа от 1 до 1 млн посредством `ReentrantLock` можно следующим образом:

```
public class CounterWithLock {
    private static final Lock lock = new ReentrantLock();

    private static int count;

    public void counter() {
        lock.lock();

        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
}
```

И давайте применим его посредством нескольких нитей исполнения:

```
CounterWithLock counterWithLock = new CounterWithLock();
Runnable task = () -> {
    counterWithLock.counter();
};
```

```
ExecutorService executor = Executors.newFixedThreadPool(8);
for (int i = 0; i < 1_000_000; i++) {
    executor.execute(task);
}
```

Готово!

В качестве бонуса посмотрим, как следующий ниже фрагмент кода представляет идиому решения задач на основе метода `ReentrantLock.lockInterruptibly()`. Исходный код, прилагаемый к этой книге, содержит пример использования метода `lockInterruptibly()`:

```
Lock / ReentrantLock lock = new ReentrantLock();

public void execute() throws InterruptedException {
    lock.lockInterruptibly();

    try {
        // сделать что-то
    } finally {
        lock.unlock();
    }
}
```

Если нить исполнения, удерживающая этот замок, прерывается, то выбрасывается исключение `InterruptedException`. При использовании метода `lock()` вместо `lockInterruptibly()` код не будет восприимчивым к прерыванию.

В дополнение к этому следующий ниже код представляет идиому использования метода `ReentrantLock.tryLock(long timeout, TimeUnit unit) throws InterruptedException`. Исходный код, прилагаемый к этой книге, также содержит пример:

```
Lock / ReentrantLock lock = new ReentrantLock();

public boolean execute() throws InterruptedException {
    if (!lock.tryLock(n, TimeUnit.SECONDS)) {
        return false;
    }

    try {
        // сделать что-то
    } finally {
        lock.unlock();
    }

    return true;
}
```

Обратите внимание, что метод `tryLock()` пытается приобрести замок в течение указанного времени. Если это время истекает, то замок не будет приобретен нитью.

Метод не повторяет попытку автоматически. Если во время попытки приобрести замок нить прерывается, будет выброшено исключение `InterruptedException`.

Наконец, исходный код, прилагаемый к этой книге, содержит пример использования метода `ReentrantLock.newCondition()`. Идиома приведена на рис. 11.6.

newCondition

```
Lock/ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition();

public void execute() throws InterruptedException {
    lock.lock();
    try {
        ...
        while/if(some_condition) {
            condition.await();
        }
    } finally {
        lock.unlock();
    }
}

lock.lock();
try {
    condition.signal(All)();
} finally {
    lock.unlock();
}
```

Когда вызывается метод `await()`, нить исполнения освобождает замок. После получения сигнала "продолжить" эта нить должна приобрести замок снова.

Рис. 11.6

223. Класс *ReentrantReadWriteLock*

В типичной ситуации тандем "чтение–запись" (например, чтение–запись файла) должен осуществляться на основе двух инструкций:

- ◆ читатели могут читать одновременно, пока нет писателей (совместный пессимистический замок);
- ◆ один писатель может писать в каждый момент времени (эксклюзивный/пессимистический замок).

Схема на рис. 11.7 показывает читателей с левой стороны и писателей с правой стороны.

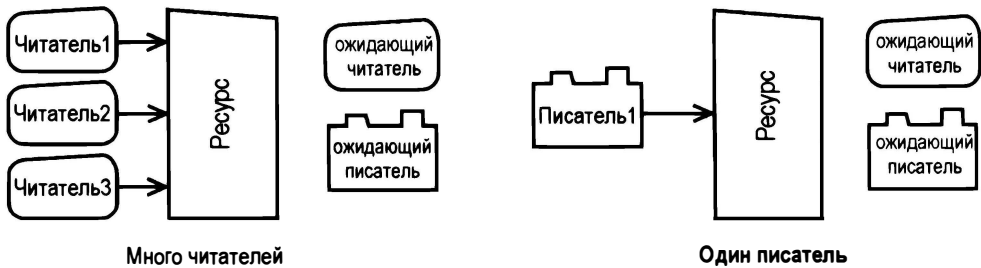


Рис 11.7

В общих чертах следующее ниже поведение имплементируется классом

`ReentrantReadWriteLock`:

- ◆ обеспечивает пессимистическую замковую семантику для обоих замков (замок на чтение и запись);
- ◆ если некоторые читатели удерживают замок на чтение, а писатель хочет приобрести замок на запись, то никакие другие читатели не могут приобрести замок на чтение до тех пор, пока писатель не освободит замок на запись;
- ◆ писатель может приобрести замок на чтение, но читатель не может приобрести замок на запись.



В случае несправедливых замков порядок, в котором нитям исполнения предоставляется доступ, не определен. Если замок должен быть справедливым (отдать приоритет нити исполнения, которая ждала дольше всего), следует использовать конструктор `ReentrantReadWriteLock(boolean fair)`.

Идиома использования класса `ReentrantReadWriteLock` такова:

```
ReadWriteLock / ReentrantReadWriteLock lock
    = new ReentrantReadWriteLock();
...
```

```
lock.readLock() / writeLock().lock();
```

```
try {
    ...
} finally {
    lock.readLock() / writeLock().unlock();
}
```

Следующий фрагмент кода представляет вариант использования класса `ReentrantReadWriteLock`, который читает и пишет в целочисленную переменную `amount`:

```
public class ReadWriteWithLock {
    private static final Logger logger
        = Logger.getLogger(ReadWriteWithLock.class.getName());
    private static final Random rnd = new Random();

    private static final ReentrantReadWriteLock lock
        = new ReentrantReadWriteLock(true);

    private static final Reader reader = new Reader();
    private static final Writer writer = new Writer();

    private static int amount;

    private static class Reader implements Runnable {
        @Override
```

```
public void run() {
    if (lock.isWriteLocked()) {
        logger.warning(() -> Thread.currentThread().getName()
            + " сообщает, что замок удерживается писателем...");
    }

    lock.readLock().lock();

    try {
        logger.info(() -> "Прочитать сумму: " + amount
            + " нитью " + Thread.currentThread().getName());
    } finally {
        lock.readLock().unlock();
    }
}

private static class Writer implements Runnable {
    @Override
    public void run() {
        lock.writeLock().lock();
        try {
            Thread.sleep(rnd.nextInt(2000));
            logger.info(() -> "Увеличить сумму на 10 нитью "
                + Thread.currentThread().getName());

            amount += 10;
        } catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
            logger.severe(() -> "Исключение: " + ex);
        } finally {
            lock.writeLock().unlock();
        }
    }
    ...
}
```

И давайте выполним 10 чтений и 10 записей с двумя читателями и четырьмя писателями:

```
ExecutorService readerService = Executors.newFixedThreadPool(2);
ExecutorService writerService = Executors.newFixedThreadPool(4);

for (int i = 0; i < 10; i++) {
    readerService.execute(reader);
    writerService.execute(writer);
}
```

Возможный результат будет таким:

```
[09:09:25] [INFO] Прочитать сумму: 0 нитью pool-1-thread-1
[09:09:25] [INFO] Прочитать сумму: 0 нитью pool-1-thread-2
[09:09:26] [INFO] Увеличить сумму на 10 нитью pool-2-thread-1
[09:09:27] [INFO] Увеличить сумму на 10 нитью pool-2-thread-2
[09:09:28] [INFO] Увеличить сумму на 10 нитью pool-2-thread-4
[09:09:29] [INFO] Увеличить сумму на 10 нитью pool-2-thread-3
[09:09:29] [INFO] Прочитать сумму: 40 нитью pool-1-thread-2
[09:09:29] [INFO] Прочитать сумму: 40 нитью pool-1-thread-1
[09:09:31] [INFO] Увеличить сумму на 10 нитью pool-2-thread-1
...
```



Прежде чем принять решение опираться на класс `ReentrantReadWriteLock`, пожалуйста, учтите тот факт, что он может страдать от голода (например, когда писатели получают приоритет, читатели могут голодать). Кроме того, мы не смогли обновить замок на чтение до замка на запись (возможен понижающий переход от писателя к читателю), и нет поддержки оптимистичного чтения. Если что-то из этого имеет для вас значение, то обратите внимание на класс `StampedLock`, который мы рассмотрим в следующей задаче.

224. Класс *StampedLock*

Класс `StampedLock` работает лучше, чем класс `ReentrantReadWriteLock` и поддерживает оптимистические чтения. Он не похож на замок с повторным входом, поэтому он склонен к тупикам. Приобретение замка возвращает метку (значение типа `long`), которая используется в блоке `finally` для снятия замка. Каждая попытка приобрести замок приводит к новой метке, и если замок не доступен, он может блокировать выполнение до тех пор, пока не станет доступным. Другими словами, если текущая нить исполнения удерживает замок и пытается приобрести замок снова, то это может привести к тупику.

Процесс оркестровки чтения/записи в классе `StampedLock` достигается посредством нескольких методов следующим образом.

- ◆ Метод `readLock()` неэксклюзивно приоретает замок, блокируя нить при необходимости до тех пор, пока он не будет доступен. Для неблокирующей попытки приобрести замок на чтение у нас есть метод `tryReadLock()`. Для блокирования по таймауту у нас есть метод `tryReadLock(long time, TimeUnit unit)`. Возвращаемая метка используется в методе `unlockRead()`.
- ◆ Метод `writeLock()` эксклюзивно приоретает замок, блокируя обработку при необходимости до тех пор, пока он не будет доступен. Для неблокирующей попытки приобрести замок на запись у нас есть метод `tryWriteLock()`. Для блокирования по таймауту у нас есть метод `tryWriteLock(long time, TimeUnit unit)`. Возвращаемая метка используется в методе `unlockWrite()`.

- ◆ Метод `tryOptimisticRead()` добавляет большой плюс к классу `StampedLock`. Этот метод возвращает метку, которая должна быть проверена посредством флагового метода `validate()`. Если замок в данный момент не удерживается в режиме записи, то возвращаемая метка не равна нулю.

Идиомы для методов `readLock()` и `writeLock()` довольно прямолинейны:

```
StampedLock lock = new StampedLock();
...
long stamp = lock.readLock() / writeLock();

try {
    ...
} finally {
    lock.unlockRead(stamp) / unlockWrite(stamp);
}
```

Попытка дать идиому для метода `tryOptimisticRead()` может привести к следующему:

```
StampedLock lock = new StampedLock();

int x; // нить-писатель может модифицировать x
...
long stamp = lock.tryOptimisticRead();
int thex = x;

if (!lock.validate(stamp)) {
    stamp = lock.readLock();

    try {
        thex = x;
    } finally {
        lock.unlockRead(stamp);
    }
}

return thex;
```

В этой идиоме обратите внимание на то, что начальное значение (x) назначается переменной `thex` после получения оптимистического замка на чтение. Затем флаговый метод `validate()` используется для проверки того, что помеченный замок не был приобретен эксклюзивно с момента эмитирования данной метки. Если метод `validate()` возвращает `false` (эквивалентно тому факту, что замок на запись приобретается нитью исполнения после того, как оптимистический замок был приобретен), то замок на чтение приобретается посредством блокирующего метода `readLock()`, и значение (x) назначается снова. Имейте в виду, что если существует любой замок на запись, то замок на чтение может блокировать. Приобретение оп-

тимистического замка позволяет нам читать одно или несколько значений и затем проводить проверку на предмет того, имеется ли какое-либо изменение в этом значении (значениях). Нам придется пройти через блокирующий замок на чтение, только если он есть.

Следующий ниже исходный код представляет вариант использования класса `StampedLock`, который читает и пишет в целочисленную переменную `amount`. В сущности, мы повторяем решение предыдущей задачи посредством оптимистических чтений:

```
public class ReadWriteWithStampedLock {
    private static final Logger logger
        = Logger.getLogger(ReadWriteWithStampedLock.class.getName());
    private static final Random rnd = new Random();

    private static final StampedLock lock = new StampedLock();

    private static final OptimisticReader optimisticReader
        = new OptimisticReader();

    private static final Writer writer = new Writer();

    private static int amount;

    private static class OptimisticReader implements Runnable {
        @Override
        public void run() {
            long stamp = lock.tryOptimisticRead();

            // если штамп для tryOptimisticRead() не действителен,
            // то нить исполнения пытается приобрести замок на чтение
            if (!lock.validate(stamp)) {
                stamp = lock.readLock();
                try {
                    logger.info(() -> "Прочитать сумму (замок на чтение): " + amount
                        + " нитью " + Thread.currentThread().getName());
                } finally {
                    lock.unlockRead(stamp);
                }
            } else {
                logger.info(() -> "Прочитать сумму (оптимистическое чтение): " + amount
                    + " нитью " + Thread.currentThread().getName());
            }
        }
    }
}
```



```
private static class Writer implements Runnable {
    @Override
    public void run() {
        long stamp = lock.writeLock();

        try {
            Thread.sleep(rnd.nextInt(2000));
            logger.info(() -> "Увеличить сумму на 10 нитью "
                + Thread.currentThread().getName());

            amount += 10;
        } catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
            logger.severe(() -> "Исключение: " + ex);
        } finally {
            lock.unlockWrite(stamp);
        }
    }
}
...
}
```

И давайте выполним 10 чтений и 10 записей с двумя читателями и четырьмя писателями:

```
ExecutorService readerService = Executors.newFixedThreadPool(2);
ExecutorService writerService = Executors.newFixedThreadPool(4);

for (int i = 0; i < 10; i++) {
    readerService.execute(optimisticReader);
    writerService.execute(writer);
}
```

Возможный результат будет следующим:

```
...
[12:12:07] [INFO] Увеличить сумму на 10 нитью pool-2-thread-4
[12:12:07] [INFO] Прочитать сумму (замок на чтение): 90 нитью pool-1-thread-2
[12:12:07] [INFO] Прочитать сумму (оптимистическое чтение): 90 нитью pool-1-thread-2
[12:12:07] [INFO] Увеличить сумму на 10 нитью pool-2-thread-1
...
```

Начиная с JDK 10, мы можем запрашивать тип метки, используя методы `isWriteLockStamp()`, `isReadLockStamp()`, `isLockStamp()` и `isOptimisticReadStamp()`. Основываясь на этом типе, мы можем выбирать надлежащий метод снятия замка, например следующим образом:

```
if (StampedLock.isReadLockStamp(stamp))
    lock.unlockRead(stamp);
}
```

В исходном коде, прилагаемом к этой книге, также есть приложение для демонстрации метода `tryConvertToWriteLock()`. В дополнение к этому, вас, возможно, заинтересует разработка приложений, использующих методы `tryConvertToReadLock()` и `tryConvertToOptimisticRead()`.

225. Тупик (обед философов)

Что такое тупик (англ. *deadlock* — смертельное объятие, мертвая хватка)? Известный анекдот из Интернета объясняет его следующим образом:

Интервьюер: объясните нам понятие тупика, и мы вас найдем!

Я: наймите меня, и я вам все объясню...

Простой тупик можно объяснить как нить А, удерживающая замок L и пытающаяся приобрести замок P; и в то же самое время имеется нить B, удерживающий замок P и пытающаяся приобрести замок L. Этот вид тупика именуется *циклическим ожиданием*. Среда Java не имеет механизма обнаружения и урегулирования тупиков (как у баз данных), и поэтому тупик бывает очень неудобным для приложения. Тупик способен полностью или частично блокировать приложение, может становиться причиной серьезного падения производительности, странного поведения и т. д. В типичной ситуации тупики трудно отлаживать, и единственный способ урегулировать тупик состоит в перезапуске приложения и надежде на лучшее.

Обед философов — это знаменитая задача, используемая для иллюстрации тупика. Итак, пять философов сидят за одним столом. Каждый из них попеременно размышляет и кушает. Для того чтобы кушать, философу нужны две вилки в руках — вилка с левой стороны и вилка с правой стороны. Трудность заключается в том, что вилок всего пять штук. Покушав, философ кладет обе вилки обратно на стол, и затем их может взять другой философ, который повторяет тот же цикл. Когда философ не кушает, он размышляет. Этот сценарий проиллюстрирован на рис. 11.8.

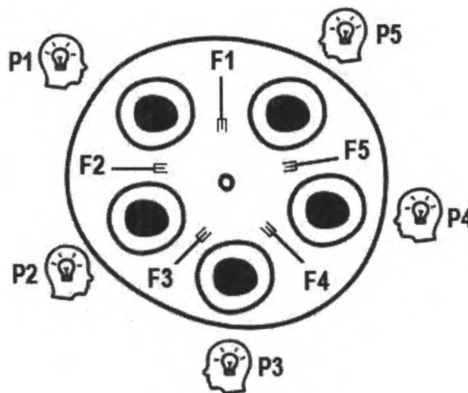


Рис. 11.8

Главная цель состоит в том, чтобы найти решение этой задачи, которое позволит философам размышлять и кушать таким образом, чтобы избежать голодной смерти.

В исходном коде мы можем рассматривать каждого философа как экземпляр `Runnable`. Будучи экземплярами `Runnable`, мы можем исполнять их в отдельных нитях. Каждый философ может взять две вилки, расположенные слева и справа от него. Если мы представляем вилку в виде значения типа `String`, то можем использовать следующий фрагмент кода:

```
public class Philosopher implements Runnable {
    private final String leftFork;
    private final String rightFork;

    public Philosopher(String leftFork, String rightFork) {
        this.leftFork = leftFork;
        this.rightFork = rightFork;
    }

    @Override
    public void run() {
        // имплементировано ниже
    }
}
```

Таким образом, философ может взять левую и правую вилки. Но поскольку все присутствующие за столом делят эти вилки между собой, то философ должен приобрести эксклюзивные замки на эти две вилки. Иметь эксклюзивный замок на левой и эксклюзивный замок на правой вилах равносильно тому, чтобы иметь в руках две вилки. Наличие эксклюзивных замков на левой и правой вилках равносильно тому, что философ кушает. Освобождение обоих эксклюзивных замков равносильно тому, что философ не кушает, а размышляет.

Приобретение замка достигается посредством ключевого слова `synchronized`, как в следующем ниже методе `run()`:

```
@Override
public void run() {
    while (true) {
        logger.info(() -> Thread.currentThread().getName() + ": размышляет");
        doIt();

        synchronized(leftFork) {
            logger.info(() -> Thread.currentThread().getName()
                + ": взял левую вилку (" + leftFork + ")");
            doIt();

            synchronized(rightFork) {
                logger.info(() -> Thread.currentThread().getName()
```

```
        + ": взял правую вилку (" + rightFork + ") and eating");
    doIt();

    logger.info(() -> Thread.currentThread().getName()
        + ": положил правую вилку ( " + rightFork + ") на стол");
    doIt();
}

logger.info(() -> Thread.currentThread().getName()
    + ": положил левую вилку (" + leftFork
    + ") на стол и размышляет");
doIt();
}
}
}
```

Философ начинает с размышления. Через некоторое время он проголодался и поэтому пытается подобрать левую и правую вилки. В случае успеха он некоторое время будет кушать. Потом он кладет вилки на стол и продолжает размышлять до тех пор, пока снова не проголодается. Тем временем другой философ будет кушать.

Метод `doIt()` симулирует соответствующие действия (размышление, прием пищи, взятие вилок со стола и возврат их на стол) посредством случайного сна. Это в коде выражено следующим образом:

```
private static void doIt() {
    try {
        Thread.sleep(rnd.nextInt(2000));
    } catch (InterruptedException ex) {
        Thread.currentThread().interrupt();
        logger.severe(() -> "Исключение: " + ex);
    }
}
```

Наконец, нам нужны вилки и философы; взгляните на следующий фрагмент кода:

```
String[] forks = {
    "Fork-1", "Fork-2", "Fork-3", "Fork-4", "Fork-5"
};

Philosopher[] philosophers = {
    new Philosopher(forks[0], forks[1]),
    new Philosopher(forks[1], forks[2]),
    new Philosopher(forks[2], forks[3]),
    new Philosopher(forks[3], forks[4]),
    new Philosopher(forks[4], forks[0])
};
```

Каждый философ будет работать в нити исполнения так:

```
Thread threadPhilosopher1 = new Thread(philosophers[0], "Philosopher-1");
...
Thread threadPhilosopher5 = new Thread(philosophers[4], "Philosopher-5");
threadPhilosopher1.start();
...
threadPhilosopher5.start();
```

Эта имплементация выглядит в полном порядке и даже может работать нормально в течение некоторого времени. Однако рано или поздно эта имплементация блокируется с результатом, который выглядит следующим образом:

```
[17:29:21] [INFO] Philosopher-5: взял левую вилку (Fork-5)
...
// ничего не происходит
```

Это тупик! Каждый философ держит в руке свою левую вилку (эксклюзивный замок на ней) и ждет, когда правая вилка окажется на столе (замок будет освобожден). Очевидно, что это ожидание не может быть удовлетворено, т. к. имеется только пять вилок, и каждый философ держит одну в своих руках.

Во избежание этого тупика есть довольно простое решение. Мы просто заставляем одного из философов сначала брать правую вилку. После успешного подбора правой вилки, он может попытаться подобрать левую. В исходном коде это сводится к быстрой модификации следующей строки:

```
// исходная строка кода
new Philosopher(forks[4], forks[0])

// модифицированная строка кода, устраняющая тупик
new Philosopher(forks[0], forks[4])
```

На этот раз мы можем выполнить приложение без тупиков.

Резюме

Что ж, вот и все! В настоящей главе были охвачены задачи, касающиеся каркаса разветвления/соединения, классов `CompletableFuture`, `ReentrantLock`, `ReentrantReadWriteLock`, `StampedLock`, атомарных переменных, отмены операции, прерываемых методов, переменных типа `ThreadLocal` и тупиков.

Вы можете скачать приложения из этой главы, чтобы увидеть результаты и получить дополнительные детали, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

12

Класс *Optional*

Эта глава содержит 24 задачи, призванные привлечь ваше внимание к нескольким правилам работы с объектом `Optional`. Проблемы и решения, представленные в этой главе, основаны на определении, данном архитектором языка Java Брайаном Гетцем:

"Класс `Optional` предназначен для обеспечения ограниченного механизма для типов, возвращаемых из библиотечных методов, где должен быть ясный способ представления результата и где использование `null` для таких методов с подавляющей вероятностью будет становиться причиной ошибок".

Но там, где есть правила, есть и исключения. Поэтому не следует делать вывод о том, что правила (или практические решения), представленные здесь, должны соблюдаться (или избегаться) любой ценой. Как всегда, все зависит от задачи, и вы должны оценивать ситуацию, взвесив все "за" и "против".

Вы также можете обратиться к плагину CDI (<https://github.com/Pscheidl/FortEE>) для Java EE (Jakarta EE), разработанной Павлом Пшейдлом (Pavel Pscheid). Он представляет собой отказоустойчивый ограничитель для Jakarta EE/Java EE, задействующий возможности шаблона `Optional`. Его мощь заключается в его простоте.

Задачи

Используйте следующие задачи для проверки вашего умения программировать класс `Optional`. Перед тем как обратиться к решениям и скачать примеры программ, настоятельно рекомендуется изучить каждую задачу.

226. **Инициализация объекта класса `Optional`.** Написать программу, которая иллюстрирует правильные и неправильные подходы для инициализации объекта класса `Optional`.
227. **Метод `Optional.get()` и недостающее значение.** Написать программу, которая иллюстрирует правильное и неправильное использование метода `Optional.get()`.
228. **Возврат уже сконструированного значения по умолчанию.** Написать программу, которая при отсутствии значения устанавливает (или возвращает)

уже сконструированное и заданное по умолчанию значение посредством метода `Optional.orElse()`.

- 229. **Возврат несуществующего значения по умолчанию.** Написать программу, которая при отсутствии значения устанавливает (или возвращает) несуществующее и заданное по умолчанию значение посредством метода `Optional.orElseGet()`.
- 230. **Выбрасывание исключения `NoSuchElementException`.** Написать программу, которая при отсутствии значения выбрасывает исключение типа `NoSuchElementException` либо другое исключение.
- 231. **Объект класса `Optional` и ссылки `null`.** Написать программу, которая иллюстрирует правильное использование метода `Optional.orElse(null)`.
- 232. **Потребление присутствующего объекта класса `Optional`.** Написать программу, которая потребляет присутствующий класс `Optional` посредством методов `ifPresent()` и `ifPresentElse()`.
- 233. **Возврат присутствующего объекта класса `Optional` или еще одного.** Допустим, что у нас есть объект класса `Optional`. Написать программу, которая опирается на метод `Optional.or()` для возврата этого `Optional` (если его значение присутствует) либо еще одного класса `Optional` (если его значение отсутствует).
- 234. **Выстраивание лямбда-выражений в цепочку посредством метода `orElseFoo()`.** Написать программу, которая иллюстрирует использование методов `orElse()` и `orElseFoo()` для предотвращения нарушения цепочек лямбда-выражений.
- 235. **Нежелательное использование типа `Optional` только для получения значения.** Привести пример плохого практического решения по выстраиванию методов `Optional` с единственной целью получения некоторых значений.
- 236. **Нежелательное использование типа `Optional` для полей.** Привести пример плохого практического решения по объявлению полей типа `Optional`.
- 237. **Нежелательное использование типа `Optional` в аргументах конструктора.** Привести пример плохого практического решения по использованию `Optional` в аргументах конструктора.
- 238. **Нежелательное использование типа `Optional` в аргументах сеттера.** Привести пример плохого практического решения по использованию `Optional` в аргументах сеттеров.
- 239. **Нежелательное использование типа `Optional` в аргументах метода.** Привести пример плохого практического решения по использованию `Optional` в аргументах методов.
- 240. **Нежелательное использование типа `Optional` для возврата пустых либо `null`-коллекций или массивов.** Привести пример плохого практического

решения по использованию Optional для возврата пустых или null-коллекций или массивов.

241. **Как избежать использования типа Optional в коллекциях.** Использование Optional в коллекциях может быть признаком плохого архитектурного дизайна. Привести пример типичного варианта использования и возможные альтернативы для избежания типа Optional в коллекциях.
242. **Путаница метода of() с методом ofNullable().** Привести пример потенциальных последствий из-за путаницы между методами Optional.of() и ofNullable().
243. **Optional<T> против OptionalInt.** Привести пример использования необобщенного OptionalInt вместо Optional<T>.
244. **Подтверждение эквивалентности объектов Optional.** Привести пример подтверждения эквивалентности объектов Optional.
245. **Преобразование значений посредством методов map() и flatMap().** Написать несколько фрагментов кода для иллюстрации использования методов Optional.map() и flatMap().
246. **Фильтрация значений посредством метода Optional.filter().** Привести пример использования метода Optional.filter() для отклонения обернутых значений, опираясь на predefined правило.
247. **Выстраивание API Optional и API потоков в цепочку.** Привести пример использования метода Optional.stream() для выстраивания в цепочку API Optional вместе с API потоков.
248. **Класс Optional и операции, чувствительные к идентичности.** Написать фрагмент кода, подтверждающий, что в случае Optional следует избегать операций, чувствительных к идентичности.
249. **Возвращение значения типа boolean при пустом объекте класса Optional.** Написать два фрагмента кода, иллюстрирующих два варианта решения задачи возвращения значения типа boolean, если заданный класс Optional является пустым.

Решения

В следующих разделах описаны решения перечисленных выше задач. Помните, что обычно не существует одного единственно правильного варианта решения той или иной задачи. Кроме того, имейте в виду, что приведенные здесь объяснения включают только самые интересные и важные детали, необходимые для решения. Вы можете скачать примеры решений, чтобы увидеть дополнительные детали и поэкспериментировать с программами, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

226. Инициализация объекта класса *Optional*

Инициализация объекта класса `Optional` должна осуществляться посредством метода `Optional.empty()` вместо `null`:

```
// Нежелательно
Optional<Book> book = null;
```

```
// Предпочтительно
Optional<Book> book = Optional.empty();
```

Поскольку объект класса `Optional` выступает в качестве контейнера (упаковки), нет смысла инициализировать его с помощью `null`.

227. Метод *Optional.get()* и недостающее значение

Итак, если мы решили вызвать метод `Optional.get()`, чтобы получить значение, возвращаемое в `Optional`, мы не должны делать это следующим образом:

```
Optional<Book> book = ...; // эта переменная может быть пустой
```

```
// Нежелательно.
// Если "book" является пустым, то следующий ниже код
// выбросит исключение java.util.NoSuchElementException
Book theBook = book.get();
```

Другими словами, прежде чем выбрать значение посредством метода `Optional.get()`, нам нужно доказать, что значение присутствует. Решение состоит из вызова метода `isPresent()` перед вызовом метода `get()`. Благодаря этому мы добавляем проверку, которая позволяет нам обрабатывать недостающее значение:

```
Optional<Book> book = ...; // эта переменная может быть пустой
```

```
// Предпочтительно
if (book.isPresent()) {
    Book theBook = book.get();
    ... // сделать что-то с "theBook"
} else {
    ... // сделать что-то, что не вызывает book.get()
}
```



Тем не менее имейте в виду, что связка из `isPresent()` и `get()` имеет плохую репутацию, и поэтому применяйте ее с осторожностью. Рассмотрите возможность использования следующих далее задач, которые предоставляют альтернативы этой связке. Учтите, что однажды метод `Optional.get()`, скорее всего, устареет.

228. Возврат уже сконструированного значения по умолчанию

Допустим, что у нас есть метод, который возвращает результат, основанный на `Optional`. Если `Optional` является пустым, то этот метод возвращает значение по умолчанию. Если рассматривать предыдущую задачу, то ее решение можно написать, например, так:

```
public static final String BOOK_STATUS = "UNKNOWN";
...
// Нежелательно
public String findStatus() {
    Optional<String> status = ...; // эта переменная может быть пустой

    if (status.isPresent()) {
        return status.get();
    } else {
        return BOOK_STATUS;
    }
}
```

Что ж, это решение неплохое, но не очень изящное. Более лаконичное и элегантное решение будет опираться на метод `Optional.orElse()`. Этот метод полезен для замены пары методов — `isPresent()` и `get()`, когда мы хотим установить или вернуть значение по умолчанию в случае пустого объекта класса `Optional`. Предыдущий фрагмент кода можно переписать следующим образом:

```
public static final String BOOK_STATUS = "UNKNOWN";
...
// Предпочтительно
public String findStatus() {
    Optional<String> status = ...; // эта переменная может быть пустой

    return status.orElse(BOOK_STATUS);
}
```



TIP

Но имейте в виду, что метод `orElse()` вычисляется даже тогда, когда объект класса `Optional` не пуст. Другими словами, метод `orElse()` вычисляется, даже если его значение не используется. С учетом сказанного, целесообразно опираться на метод `orElse()` только тогда, когда его аргумент является уже сконструированным значением. Благодаря этому мы снижаем потенциальный штраф на производительность. Следующая далее задача касается случая, когда метод `orElse()` не является правильным вариантом выбора.

229. Возврат несуществующего значения по умолчанию

Допустим, что у нас есть метод, который возвращает результат, основываясь на объекте класса `Optional`. Если этот объект класса `Optional` является пустым, то указанный метод возвращает вычисленное значение. Метод `computeStatus()` вычисляет это значение:

```
private String computeStatus() {  
    // некий код для вычисления статуса  
}
```

Неуклюжее решение будет опираться на пару методов `isPresent()` и `get()`, как показано ниже:

```
// Нежелательно  
public String findStatus() {  
    Optional<String> status = ...; // эта переменная может быть пустой  
  
    if (status.isPresent()) {  
        return status.get();  
    } else {  
        return computeStatus();  
    }  
}
```

Даже если это решение является неуклюжим, оно все равно лучше того, которое опирается на метод `orElse()`:

```
// Нежелательно  
public String findStatus() {  
    Optional<String> status = ...; // эта переменная может быть пустой  
  
    // computeStatus() вызывается, даже если "status" не является пустым  
    return status.orElse(computeStatus());  
}
```

В этом случае предпочтительное решение опирается на метод `Optional.orElseGet()`. Аргументом этого метода является поставщик `Supplier`, и поэтому он выполняется только тогда, когда значение типа `Optional` отсутствует. Этот метод намного лучше, чем метод `orElse()`, т. к. он избавляет нас от выполнения дополнительного кода, который не должен выполняться при наличии значения типа `Optional`. Таким образом, предпочтительное решение заключается в следующем:

```
// Предпочтительно  
public String findStatus() {  
    Optional<String> status = ...; // эта переменная может быть пустой  
  
    // computeStatus() вызывается, только если "status" является пустым  
    return status.orElseGet(this::computeStatus);  
}
```

230. Выбрасывание исключения *NoSuchElementException*

Иногда, если объект класса `Optional` является пустым, мы хотим выбросить исключение (например, исключение `NoSuchElementException`). Неуклюжее решение этой задачи приведено ниже:

```
// Нежелательно
public String findStatus() {
    Optional<String> status = ...; // эта переменная может быть пустой

    if (status.isPresent()) {
        return status.get();
    } else {
        throw new NoSuchElementException("Не получается найти статус");
    }
}
```

Но гораздо более элегантное решение будет опираться на метод `Optional.orElseThrow()`. Сигнатура этого метода, `orElseThrow(Supplier<? extends X>exceptionSupplier)`, позволяет нам выбрасывать исключение следующим образом (если значение присутствует, то метод `orElseThrow()` вернет его):

```
// Предпочтительно
public String findStatus() {
    Optional<String> status = ...; // эта переменная может быть пустой

    return status.orElseThrow(
        () -> new NoSuchElementException("Не получается найти статус"));
}
```

Как вариант, еще одним исключением является, например, `IllegalStateException`:

```
// Предпочтительно
public String findStatus() {
    Optional<String> status = ...; // эта переменная может быть пустой

    return status.orElseThrow(
        () -> new IllegalStateException("Не получается найти статус"));
}
```

Начиная с **JDK 10**, класс `Optional` обогащен разновидностью метода `orElseThrow()` без аргументов. Этот метод неявно выбрасывает исключение `NoSuchElementException`:

```
// Предпочтительно (JDK 10+)
public String findStatus() {
    Optional<String> status = ...; // эта переменная может быть пустой

    return status.orElseThrow();
}
```

Тем не менее имейте в виду, что выбрасывание непроверенного исключения без осмысленного сообщения в производственной среде не является хорошим практическим решением.

231. Объект класса *Optional* и ссылки *null*

Можно воспользоваться преимуществом метода `orElse(null)`, вызвав метод, который в некоторых ситуациях принимает ссылки `null`.

Кандидатом на этот сценарий является метод `Method.invoke()` из API рефлексии Java (см. главу 7).

Первый аргумент метода `Method.invoke()` представляет объектный экземпляр, на котором должен быть вызван этот конкретный метод. Если метод является статическим, то первый аргумент должен быть `null`, и поэтому нет необходимости иметь экземпляр объекта.

Допустим, что у нас есть класс с именем `Book` и вспомогательный метод, приведенный ниже.

Этот метод возвращает пустой объект класса `Optional` (если данный метод является статическим) или объект класса `Optional`, содержащий экземпляр класса `Book` (если данный метод не является статическим):

```
private static Optional<Book> fetchBookInstance(Method method) {
    if (Modifier.isStatic(method.getModifiers())) {
        return Optional.empty();
    }

    return Optional.of(new Book());
}
```

Вызвать этот метод довольно просто:

```
Method method = Book.class.getDeclaredMethod(...);
```

```
Optional<Book> bookInstance = fetchBookInstance(method);
```

Далее, если объект класса `Optional` является пустым (это означает, что метод является статическим), то мы должны передать `null` методу `Method.invoke()`; в противном случае мы передадим экземпляр класса `Book`. Неуклюжее решение может основываться на паре методов `isPresent()` и `get()` следующим образом:

```
// Нежелательно
if (bookInstance.isPresent()) {
    method.invoke(bookInstance.get());
} else {
    method.invoke(null);
}
```

Но это идеально подходит для метода `Optional.orElse(null)`. Следующий фрагмент сводит решение к одной строке кода:

```
// Предпочтительно
method.invoke(bookInstance.orElse(null));
```



В качестве общего правила, мы должны использовать метод `orElse(null)` только тогда, когда у нас есть `Optional` и нам нужна ссылка `null`. В противном случае избегайте метода `orElse(null)`.

232. Потребление присутствующего объекта класса *Optional*

Иногда мы хотим лишь потребить присутствующий объект класса `Optional`. Если `Optional` отсутствует, то ничего не нужно делать. Неуклюжее решение будет опираться на пару методов `isPresent()` и `get()` следующим образом:

```
// Нежелательно
public void displayStatus() {
    Optional<String> status = ...; // эта переменная может быть пустой

    if (status.isPresent()) {
        System.out.println(status.get());
    }
}
```

Оптимальное решение опирается на метод `ifPresent()`, который берет потребителя `Consumer` в качестве аргумента. Он является альтернативным вариантом паре методов `isPresent()` и `get()`, когда нам просто нужно потребить присутствующее значение. Исходный код можно переписать следующим образом:

```
// Предпочтительно
public void displayStatus() {
    Optional<String> status = ...; // эта переменная может быть пустой

    status.ifPresent(System.out::println);
}
```

Но в других случаях, если `Optional` отсутствует, нам нужно исполнить действие на основе пустого значения. Решение, основанное на паре методов `isPresent()` и `get()`, выглядит следующим образом:

```
// Нежелательно
public void displayStatus() {
    Optional<String> status = ...; // эта переменная может быть пустой

    if (status.isPresent()) {
        System.out.println(status.get());
    }
}
```

```
} else {  
    System.out.println("Статус не найден...");  
}  
}
```

Опять же, этот вариант выбора не лучший. В качестве альтернативы мы можем рассчитывать на метод `ifPresentOrElse()`. Этот метод доступен, начиная с JDK 9, и подобен методу `ifPresent()`; единственная разница заключается в том, что он также охватывает ветвь `else`:

```
// Предпочтительно  
public void displayStatus() {  
    Optional<String> status = ...; // эта переменная может быть пустой  
  
    status.ifPresentOrElse(System.out::println,  
        () -> System.out.println("Статус не найден..."));  
}
```

233. Возврат присутствующего объекта класса *Optional* или еще одного

Рассмотрим метод, возвращающий объект класса `Optional`. В общих чертах этот метод вычисляет объект класса `Optional` и, если он не является пустым, просто возвращает этот объект класса `Optional`. В противном случае, если вычисляемый объект класса `Optional` пуст, мы исполняем некоторое другое действие, которое также возвращает объект класса `Optional`.

Пара методов — `isPresent()` и `get()` — может сделать это следующим образом (нежелательный вариант решения):

```
private final static String BOOK_STATUS = "UNKNOWN";  
...  
// Нежелательно  
public Optional<String> findStatus() {  
    Optional<String> status = ...; // эта переменная может быть пустой  
  
    if (status.isPresent()) {  
        return status;  
    } else {  
        return Optional.of(BOOK_STATUS);  
    }  
}
```

Однако мы должны избегать таких конструкций, как следующие две:

```
return Optional.of(status.orElse(BOOK_STATUS));  
return Optional.of(status.orElseGet(() -> (BOOK_STATUS)));
```

Лучшее решение доступно, начиная с JDK 9, и оно состоит из метода `Optional.or()`. Этот метод способен возвращать объект класса `Optional`, описывающий значение.

В противном случае он возвращает объект класса `Optional`, произведенный заданной функцией `Supplier` (поставляющей функцией, которая производит возвращаемый объект класса `Optional`):

```
private final static String BOOK_STATUS = "UNKNOWN";
...
// Предпочтительно
public Optional<String> findStatus() {
    Optional<String> status = ...; // эта переменная может быть пустой

    return status.or(() -> Optional.of(BOOK_STATUS));
}
```

234. Выстраивание лямбда-выражений в цепочку посредством метода *orElseFoo()*

Некоторые операции, специфичные для лямбда-выражений, возвращают значения типа `Optional` (например, `findFirst()`, `findAny()`, `reduce()` и т. д.). Попытка обратиться к этим объектам класса `Optional` посредством пары методов `isPresent()` и `get()` является громоздкой, потому что мы должны разрывать цепочку лямбда-выражений, добавлять некоторый условный код посредством блоков `if-else` и рассматривать возможность возобновления цепочки.

Следующий фрагмент кода демонстрирует это практическое решение:

```
private static final String NOT_FOUND = "NOT FOUND";

List<Book> books...;
...
// Нежелательно
public String findFirstCheaperBook(int price) {
    Optional<Book> book = books.stream()
        .filter(b -> b.getPrice() < price)
        .findFirst();

    if (book.isPresent()) {
        return book.get().getName();
    } else {
        return NOT_FOUND;
    }
}
```

Еще один шаг, и мы можем получить что-то вроде следующего:

```
// Нежелательно
public String findFirstCheaperBook(int price) {
    Optional<Book> book = books.stream()
```



```
.filter(b -> b.getPrice() < price)
.findFirst();

return book.map(Book::getName)
.orElse(NOT_FOUND);
}
```

Вместо пары методов `isPresent()` и `get()` разумнее вызвать метод `orElse()`. Но будет еще лучше, если мы воспользуемся методом `orElse()` (и `orElseFoo()`) непосредственно в цепочке лямбда-выражений и избежим нарушения кода:

```
private static final String NOT_FOUND = "NOT FOUND";
...
// Предпочтительно
public String findFirstCheaperBook(int price) {
    return books.stream()
        .filter(b -> b.getPrice() < price)
        .findFirst()
        .map(Book::getName)
        .orElse(NOT_FOUND);
}
```

Давайте решим еще одну задачу.

На этот раз у нас есть автор нескольких книг, и мы хотим проверить, была ли некоторая книга написана этим автором. Если наш автор не писал данную книгу, то мы хотим выбросить исключение `NoSuchElementException`.

Действительно плохое решение этой задачи будет таким:

```
// Нежелательно
public void validateAuthorOfBook(Book book) {
    if (!author.isPresent() ||
        !author.get().getBooks().contains(book)) {
        throw new NoSuchElementException();
    }
}
```

А вот использование метода `orElseThrow()` может решить задачу очень элегантно:

```
// Предпочтительно
public void validateAuthorOfBook(Book book) {
    author.filter(a -> a.getBooks().contains(book))
        .orElseThrow();
}
```

235. Нежелательное использование типа *Optional* только для получения значения

Эта задача возглавляет связку задач из категории примеров "не использовать" (примеры нежелательного кода). В этом и следующих разделах даны рекомендации того, как предотвратить чрезмерное использование типа `Optional`, и сформулирова-

ны несколько правил, которые избавят нас от многих проблем. Тем не менее у правил есть исключения. Поэтому не следует делать вывод, будто представленные тут указания следует избегать любой ценой. Как всегда, все зависит от поставленной задачи.

В случае типа `Optional` часто встречающийся сценарий предусматривает выстраивание его методов в цепочку с единственной целью получения некоторого значения.

Избегайте этого подхода на практике и опирайтесь на простой и понятный код. Другими словами, избегайте делать что-то вроде следующего фрагмента кода:

```
public static final String BOOK_STATUS = "UNKNOWN";
...
// Нежелательно
public String findStatus() {
    // получить статус, который может быть null
    String status = ...;

    return Optional.ofNullable(status).orElse(BOOK_STATUS);
}
```

И используйте простой блок `if-else` или тернарный оператор (для простых случаев):

```
// Предпочтительно
public String findStatus() {
    // получить статус, который может быть null
    String status = null;

    return status == null ? BOOK_STATUS : status;
}
```

236. Нежелательное использование типа *Optional* для полей

Категория примеров нежелательного кода продолжается следующей рекомендацией — тип `Optional` не предназначен для использования в полях и не имплементирует интерфейс `Serializable`.

Класс `Optional` определенно не предназначен для применения в качестве поля `JavaBean`, поэтому вот так делать не следует:

```
// Нежелательно
public class Book {
    [access_modifier][static][final]
        Optional<String> title;
    [access_modifier][static][final]
        Optional<String> subtitle = Optional.empty();
    ...
}
```

Но следует делать вот так:

```
// Предпочтительно
public class Book {
    [access_modifier][static][final] String title;
    [access_modifier][static][final] String subtitle = "";
    ...
}
```

237. Нежелательное использование типа *Optional* в аргументах конструктора

Категория примеров нежелательного кода продолжается еще одним сценарием. Имейте в виду, что тип `Optional` представляет собой контейнер для объектов; поэтому тип `Optional` добавляет еще один уровень абстракции. Другими словами, неправильное использование типа `Optional` просто добавляет дополнительный стереотипный код.

Взгляните на следующий ниже вариант использования объекта класса `Optional`, который хорошо это показывает (этот код нарушает предыдущий *разд. 236 "Нежелательное использование типа `Optional` для полей"*):

```
// Нежелательно
public class Book {
    // не может быть null
    private final String title;

    // опциональное поле, не может быть null
    private final Optional<String> isbn;

    public Book(String title, Optional<String> isbn) {
        this.title = Objects.requireNonNull(title,
            () -> "Заголовок не может быть null");

        if (isbn == null) {
            this.isbn = Optional.empty();
        } else {
            this.isbn = isbn;
        }
        // либо
        this.isbn = Objects.requireNonNullElse(isbn, Optional.empty());
    }

    public String getTitle() {
        return title;
    }
}
```

```

public Optional<String> getIsbn() {
    return isbn;
}
}

```

Мы можем исправить этот код, удалив опциональные поля и аргументы конструктора, как показано ниже:

```

// Предпочтительно
public class Book {
    private final String title; // не может быть null
    private final String isbn;  // может быть null

    public Book(String title, String isbn) {
        this.title = Objects.requireNonNull(title,
            () -> "Заголовок не может быть null");
        this.isbn = isbn;
    }

    public String getTitle() {
        return title;
    }

    public Optional<String> getIsbn() {
        return Optional.ofNullable(isbn);
    }
}

```

Геттер поля `isbn` возвращает тип `Optional`. Но не рассматривайте этот пример в качестве правила для преобразования всех ваших геттеров таким образом. Некоторые геттеры возвращают коллекции или массивы, и в этом случае они предпочитают возвращать пустые коллекции/массивы вместо типа `Optional`. Используйте этот технический прием, помня высказывание Брайана Гетца (архитектора языка Java):

"Я думаю, что использовать его регулярно в качестве возвращаемого значения для геттеров определенно было бы чрезмерным".

Брайан Гетц

238. Нежелательное использование типа *Optional* в аргументах сеттера

Категория примеров нежелательного кода продолжается очень заманчивым сценарием, который состоит из использования типа `Optional` в аргументах сеттера. Следующий ниже фрагмент кода следует избегать, т. к. он добавляет лишний стерео-

типный код и нарушает рекомендации из разд. 236 "*Нежелательное использование типа `Optional` для полей*" (обратите внимание на метод `setIsbn()`):

```
// Нежелательно
public class Book {
    private Optional<String> isbn;

    public Optional<String> getIsbn() {
        return isbn;
    }

    public void setIsbn(Optional<String> isbn) {
        if (isbn == null) {
            this.isbn = Optional.empty();
        } else {
            this.isbn = isbn;
        }

        // либо
        this.isbn = Objects.requireNonNullElse(isbn, Optional.empty());
    }
}
```

Мы можем исправить этот код, удалив тип `Optional` из полей и из аргументов сеттеров:

```
// Предпочтительно
public class Book {
    private String isbn;

    public Optional<String> getIsbn() {
        return Optional.ofNullable(isbn);
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
}
```



Чаще всего этот плохой практический прием применяют в сущностях JPA для постоянных свойств (для увязки атрибута сущности как тип `Optional`). Однако существует возможность использования типа `Optional` в сущностях доменной модели.

239. Нежелательное использование типа *Optional* в аргументах метода

Категория примеров нежелательного кода продолжается еще одной часто встречающейся ошибкой использования типа `Optional`. На этот раз давайте обратимся к применению типа `Optional` в аргументах метода.

Тип `Optional` в аргументах метода — это просто еще один пример использования, который может привести к излишне усложненному коду. Рекомендуется программно проверять аргументы на равенство `null` вместо того, чтобы полагаться на создание объектов класса `Optional` вызывающим кодом, в особенности пустых объектов класса `Optional`. Этот плохой практический прием загромождает код и может выбрасывать исключение `NullPointerException`. Вызывающий код по-прежнему может передавать `null`. И поэтому вы вернулись к простой проверке аргументов на `null`.

Имейте в виду, что объект класса `Optional` — это всего лишь еще один объект (контейнер) и к тому же не дешевый. Тип `Optional` потребляет в четыре раза больше памяти, чем простая ссылка!

И подумайте дважды, прежде чем делать что-то вроде следующего:

```
// Нежелательно
public void renderBook(Format format,
    Optional<Renderer> renderer, Optional<String> size) {

    Objects.requireNonNull(format, "Формат не может быть null");
    Renderer bookRenderer = renderer.orElseThrow(
        () -> new IllegalArgumentException("Отрисовщик не может быть пустым")
    );
    String bookSize = size.orElseGet(() -> "125 x 200");
    ...
}
```

Обратите внимание на следующий вызов этого метода, который создает требуемый объект класса `Optional`. Очевидно, что передача `null` также возможна и приведет к исключению `NullPointerException`, но это означает, что вы намеренно нарушаете цель типа `Optional` — не думать о засорении приведенного выше кода проверками на `null` для параметров типа `Optional`; это было бы действительно плохой идеей:

```
Book book = new Book();

// Нежелательно
book.renderBook(new Format(),
    Optional.of(new CoolRenderer()), Optional.empty());

// Нежелательно.
// Приводит к исключению NPE
book.renderBook(new Format(),
    Optional.of(new CoolRenderer()), null);
```

Мы можем исправить этот код, удалив объекты класса `Optional`:

```
// Предпочтительно
public void renderBook(Format format, Renderer renderer, String size) {
    Objects.requireNonNull(format, "Формат не может быть null");
    Objects.requireNonNull(renderer, "Отрисовщик не может быть null");
```

```
String bookSize = Objects.requireNonNullElseGet(
    size, () -> "125 x 200");
...
}
```

На этот раз вызов этого метода не заставляет создавать `Optional`:

```
Book book = new Book();
```

```
// Предпочтительно
```

```
book.renderBook(new Format(), new CoolRenderer(), null);
```



Когда метод может принимать необязательные параметры, следует опираться на классическую перегрузку метода, а не на тип `Optional`.

240. Нежелательное использование типа *Optional* для возврата пустых либо *null*-коллекций или массивов

Далее давайте рассмотрим использование типа `Optional` в качестве возвращаемого типа, который обертывает пустую либо `null`-коллекцию или массив.

Возвращение типа `Optional`, который обертывает пустую или `null`-коллекцию/массив, может состоять из чистого и легковесного кода. Взгляните на следующий код, который это показывает:

```
// Нежелательно
public Optional<List<Book>> fetchBooksByYear(int year) {
    // при получении книг может быть возвращено null
    List<Book> books = ...;

    return Optional.ofNullable(books);
}
```

```
Optional<List<Book>> books = author.fetchBooksByYear(2021);
```

```
// Нежелательно
public Optional<Book[]> fetchBooksByYear(int year) {
    // при получении книг может быть возвращено null
    Book[] books = ...;

    return Optional.ofNullable(books);
}
```

```
Optional<Book[]> books = author.fetchBooksByYear(2021);
```

Мы можем очистить этот код, удалив ненужные объекты класса `Optional`, а затем оперевшись на пустые коллекции (например, `Collections.emptyList()`, `emptyMap()` и

`emptySet()`) и массивы (например, `new String[0]`). Вот это решение является предпочтительным:

```
// Предпочтительно
public List<Book> fetchBooksByYear(int year) {
    // при получении книг может быть возвращено null
    List<Book> books = ...;

    return books == null ? Collections.emptyList() : books;
}
```

```
List<Book> books = author.fetchBooksByYear(2021);
```

```
// Предпочтительно
public Book[] fetchBooksByYear(int year) {
    // при получении книг может быть возвращено null
    Book[] books = ...;

    return books == null ? new Book[0] : books;
}
```

```
Book[] books = author.fetchBooksByYear(2021);
```

Если вам нужно проводить различие между отсутствующей и пустой коллекцией/массивом, то следует выбрасывать исключение для отсутствующей.

241. Как избежать использования типа *Optional* в коллекциях

Опора на тип `Optional` в коллекциях может быть признаком плохого архитектурного дизайна. Потратьте еще 30 минут на переоценку задачи и поиск более подходящих решений.

Предыдущее утверждение справедливо особенно в случае коллекции `Map`, когда причина этого решения звучит следующим образом: "Итак, коллекция `Map` возвращает `null`, если нет соответствия для ключа или с ключом сопоставляется `null`, поэтому я не могу сказать, присутствует ли ключ или отсутствует значение. Я оберну значения посредством метода `Optional.ofNullable()`, и дело сделано!"

Но что мы будем делать дальше, если коллекция `Map`, состоящая из `Optional<Foo>`, заполняется значениями `null`, отсутствующими опциональными значениями или даже объектами `Optional`, которые содержат что-то еще, но не `Foo`? Разве мы только что не вложили изначальную проблему в еще один слой?

Как насчет штрафа на производительность? Тип `Optional` не идет за бесплатно; это просто еще один объект, который потребляет память и должен быть собран.

Итак, давайте рассмотрим решение, которого следует избегать:

```
private static final String NOT_FOUND = "NOT FOUND";
...
```



```
// Нежелательно
Map<String, Optional<String>> isbnns = new HashMap<>();
isbnns.put("Book1", Optional.ofNullable(null));
isbnns.put("Book2", Optional.ofNullable("123-456-789"));
...
Optional<String> isbn = isbnns.get("Book1");

if (isbn == null) {
    System.out.println("Не получается найти этот ключ");
} else {
    String unwrappedIsbn = isbn.orElse(NOT_FOUND);
    System.out.println("Ключ найден, значение: " + unwrappedIsbn);
}
```

Более подходящее и элегантное решение может опираться на JDK 8 и его метод `getOrDefault()` следующим образом:

```
private static String get(Map<String, String> map, String key) {
    return map.getOrDefault(key, NOT_FOUND);
}
```

```
Map<String, String> isbnns = new HashMap<>();
isbnns.put("Book1", null);
isbnns.put("Book2", "123-456-789");
...
String isbn1 = get(isbnns, "Book1"); // null
String isbn2 = get(isbnns, "Book2"); // 123-456-789
String isbn3 = get(isbnns, "Book3"); // NOT FOUND
```

Другие решения могут опираться:

- ◆ на метод `containsKey()`;
- ◆ тривиальную имплементацию путем расширения коллекции `HashMap`;
- ◆ метод `computeIfAbsent()` JDK 8;
- ◆ коллекцию `DefaultedMap` библиотеки `Apache Commons`.

Мы можем сделать вывод, что всегда есть более подходящие решения, чем использование типа `Optional` в коллекциях.

Но рассмотренный ранее вариант использования не является наихудшим сценарием. Вот еще два приема, которых следует избегать:

```
Map<Optional<String>, String> items = new HashMap<>();
Map<Optional<String>, Optional<String>> items = new HashMap<>();
```

242. Путаница метода `of()` с методом `ofNullable()`

Перепутывание или ошибочное использование метода `Optional.of()` вместо метода `Optional.ofNullable()`, или наоборот, может привести к странному поведению и даже исключению `NullPointerException`.



Метод `Optional.of(null)` будет выбрасывать исключение `NullPointerException`, тогда как метод `Optional.ofNullable(null)` будет приводить к `Optional.empty`.

Взгляните на следующую безуспешную попытку написать фрагмент кода, чтобы избежать исключение `NullPointerException`:

```
// Нежелательно
public Optional<String> isbn(String bookId) {
    // при получении "isbn" может быть возвращено null для заданного "bookId"
    String isbn = ...;

    return Optional.of(isbn); // этот код выбрасывает исключение,
                             // если "isbn" равно null :(
}
```

Но, скорее всего, мы хотели использовать метод `ofNullable()`, как показано ниже:

```
// Предпочтительно
public Optional<String> isbn(String bookId) {
    // при получении "isbn" может быть возвращено null для заданного "bookId"
    String isbn = ...;

    return Optional.ofNullable(isbn);
}
```

Использование метода `ofNullable()` вместо метода `of()` не является катастрофой, но может вызвать некоторую путаницу и не принести никакой пользы. Взгляните на следующий код:

```
// Нежелательно.
// Метод ofNullable() не добавляет никакого значения
return Optional.ofNullable("123-456-789");

// Предпочтительно
return Optional.of("123-456-789"); // отсутствие риска исключения NPE
```

Вот еще одна задача. Допустим, что мы хотим преобразовать пустой объект типа `String` в пустой объект `Optional`. Можно подумать, что правильное решение будет опираться на метод `of()` следующим образом:

```
// Нежелательно
Optional<String> result = Optional.of(str)
    .filter(not (String::isEmpty));
```

Но помните, что объект типа `String` может быть `null`. Это решение будет хорошо работать для пустых или непустых строк, но не для `null`-строк. Поэтому метод `ofNullable()` дает нам правильное решение:

```
// Предпочтительно
Optional<String> result = Optional.ofNullable(str)
    .filter(not (String::isEmpty));
```

243. *Optional*<T> против *OptionalInt*

Если нет никакой конкретной причины для использования упакованных примитивов, то рекомендуется избегать *Optional*<T> и опираться на непривычные типы *OptionalInt*, *OptionalLong* или *OptionalDouble*.

Операции упаковки и распаковки являются дорогостоящими и могут приводить к штрафам на производительность. В устранении этого риска мы можем опереться на типы *OptionalInt*, *OptionalLong* и *OptionalDouble*. Они являются обертками для примитивных типов *int*, *long* и *double*.

Поэтому избегайте следующих ниже (и подобных им) решений:

```
// Нежелательно
Optional<Integer> priceInt = Optional.of(50);
Optional<Long> priceLong = Optional.of(50L);
Optional<Double> priceDouble = Optional.of(49.99d);
```

И отдавайте предпочтение таким решениям:

```
// Предпочтительно
// развернуть посредством getAsInt()
OptionalInt priceInt = OptionalInt.of(50);

// развернуть посредством getAsLong()
OptionalLong priceLong = OptionalLong.of(50L);

// развернуть посредством getAsDouble()
OptionalDouble priceDouble = OptionalDouble.of(49.99d);
```

244. Подтверждение эквивалентности объектов *Optional*

Наличие двух объектов *Optional* в методе *assertEquals()* не требует развернутых значений. Это работает, потому что метод *Optional.equals()* сравнивает завернутые значения, а не объекты *Optional*. Вот исходный код метода *Optional.equals()*:

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }

    if (!(obj instanceof Optional)) {
        return false;
    }

    Optional<?> other = (Optional<?>) obj;

    return Objects.equals(value, other.value);
}
```

Допустим, что у нас есть два объекта Optional:

```
Optional<String> actual = ...;
Optional<String> expected = ...;
```

```
// либо
```

```
Optional actual = ...;
Optional expected = ...;
```

Желательно избегать проверки, написанной следующим образом:

```
// Нежелательно
@Test
public void givenOptionalsWhenTestEqualityThenTrue() throws Exception {
    assertEquals(expected.get(), actual.get());
}
```

Если ожидаемое и/или фактическое значение является пустым, то метод `get()` станет причиной исключения `NoSuchElementException`.

Лучше использовать следующую ниже проверку:

```
// Предпочтительно
@Test
public void givenOptionalsWhenTestEqualityThenTrue() throws Exception {
    assertEquals(expected, actual);
}
```

245. Преобразование значений посредством методов *map()* и *flatMap()*

Методы `Optional.map()` и `flatMap()` удобны для преобразования значения объекта класса `Optional`.

Метод `map()` применяет функциональный аргумент к значению, а затем возвращает результат, завернутый в объект класса `Optional`. Метод `flatMap()` применяет функциональный аргумент к значению, а затем возвращает результат непосредственно.

Допустим, у нас есть `Optional<String>`, и мы хотим преобразовать эту строку из нижнего регистра в верхний. Скучное решение может быть написано следующим образом:

```
Optional<String> lowername = ...; // может быть пустым
```

```
// Нежелательно
```

```
Optional<String> uppername;
```

```
if (lowername.isPresent()) {
    uppername = Optional.of(lowername.get().toUpperCase());
} else {
    uppername = Optional.empty();
}
```

Более вдохновляющее решение (в одной строке кода) будет опираться на метод `Optional.map()`:

```
// Предпочтительно
```

```
Optional<String> upername = lowername.map(String::toUpperCase);
```

Метод `map()` также может быть полезен во избежание разрыва цепочки лямбда-выражений. Возьмем список `List<Book>`, в котором мы хотим отыскать первую книгу на \$50 дешевле, и если такая книга существует, то перевести ее название в верхний регистр. Опять же решение в лоб будет следующим:

```
private static final String NOT_FOUND = "NOT FOUND";
```

```
List<Book> books = Arrays.asList();
```

```
...
```

```
// Нежелательно
```

```
Optional<Book> book = books.stream()
```

```
    .filter(b -> b.getPrice() < 50)
```

```
    .findFirst();
```

```
String title;
```

```
if (book.isPresent()) {
```

```
    title = book.get().getTitle().toUpperCase();
```

```
} else {
```

```
    title = NOT_FOUND;
```

```
}
```

Опираясь на метод `map()`, мы можем сделать это посредством следующей цепочки лямбда-выражений:

```
// Предпочтительно
```

```
String title = books.stream()
```

```
    .filter(b -> b.getPrice() < 50)
```

```
    .findFirst()
```

```
    .map(Book::getTitle)
```

```
    .map(String::toUpperCase)
```

```
    .orElse(NOT_FOUND);
```

В приведенном выше примере метод `getTitle()` является классическим геттером, который возвращает название книги в качестве экземпляра типа `String`. Но давайте модифицируем этот геттер, чтобы он возвращал объект класса `Optional`:

```
public Optional<String> getTitle() {
```

```
    return ...;
```

```
}
```

На этот раз мы не можем использовать метод `map()`, потому что `map(Book::getTitle)` будет возвращать `Optional<Optional<String>>` вместо `Optional<String>`. Если же мы обопремся на метод `flatMap()`, то возвращаемое им значение не будет обернуто в дополнительный объект класса `Optional`:

```
// Предпочтительно
```

```
String title = books.stream()
```

```
.filter(b -> b.getPrice() < 50)
.findFirst()
.flatMap(Book::getTitle)
.map(String::toUpperCase)
.orElse(NOT_FOUND);
```

Таким образом, метод `Optional.map()` обертывает результат преобразования в объект класса `Optional`. Если этот результат сам по себе является объектом класса `Optional`, то мы получаем `Optional<Optional<...>>`. С другой стороны, метод `flatMap()` не обертывает результат в дополнительный объект класса `Optional`.

246. Фильтрация значений посредством метода *`Optional.filter()`*

Подход с использованием метода `Optional.filter()` для принятия или отклонения обернутого значения является очень удобным, поскольку его можно выполнять без явного развертывания значения. Мы просто передаем предикат (условие) в качестве аргумента и получаем объект класса `Optional` (исходный объект `Optional`, если условие удовлетворено, либо пустой объект класса `Optional`, если условие не удовлетворено).

Рассмотрим следующий скучный подход для проверки длины ISBN-кода книги:

```
// Нежелательно
public boolean validateIsbnLength(Book book) {
    Optional<String> isbn = book.getIsbn();

    if (isbn.isPresent()) {
        return isbn.get().length() > 10;
    }

    return false;
}
```

Приведенное решение опирается на явное развертывание значения типа `Optional`. Но если мы опираемся на метод `Optional.filter()`, то можем сделать это без явного развертывания:

```
// Предпочтительно
public boolean validateIsbnLength(Book book) {
    Optional<String> isbn = book.getIsbn();

    return isbn.filter((i) -> i.length() > 10)
               .isPresent();
}
```



Метод `Optional.filter()` также полезен для предотвращения разрыва цепочки лямбда-выражений.

247. Выстраивание API *Optional* и API потоков в цепочку

Начиная с JDK 9, мы можем ссылаться на экземпляр класса `Optional` как на `Stream`, применяя метод `Optional.stream()`.

Это очень полезно, когда мы должны выстроить в цепочку API `Optional` и API потоков. Метод `Optional.stream()` возвращает поток из одного элемента (значение объекта класса `Optional`) либо пустой поток (если объект класса `Optional` не имеет значения). Далее мы можем использовать все методы, доступные в API потоков.

Допустим, что у нас есть метод получения книг по ISBN-коду (если ни одна книга не соответствует заданному ISBN-коду, то этот метод возвращает пустой объект класса `Optional`):

```
public Optional<Book> fetchBookByIsbn(String isbn) {  
    // получение книги по заданному "isbn" может возвращать null  
    Book book = ...;  
  
    return Optional.ofNullable(book);  
}
```

В дополнение к этому мы обходим список ISBN-кодов в цикле и возвращаем список из объектов класса `Book` следующим образом (каждый ISBN-код передается через метод `fetchBookByIsbn()`):

```
// Нежелательно  
public List<Book> fetchBooks(List<String> isbnns) {  
    return isbnns.stream()  
        .map(this::fetchBookByIsbn)  
        .filter(Optional::isPresent)  
        .map(Optional::get)  
        .collect(toList());  
}
```

Здесь в центре внимания находятся следующие две строки кода:

```
.filter(Optional::isPresent)  
.map(Optional::get)
```

Поскольку метод `fetchBookByIsbn()` может возвращать пустые объекты `Optional`, мы должны обеспечить их устранение из конечного результата. Для этого мы вызываем метод `Stream.filter()` и применяем метод `Optional.isPresent()` к каждому объекту класса `Optional`, возвращаемому методом `fetchBookByIsbn()`. Таким образом, после фильтрации мы имеем только классы `Optional` с присутствующими значениями. Далее мы применяем метод `Stream.map()` для развертывания этих объектов класса `Optional` в объекты класса `Book`. Наконец, мы собираем объекты класса `Book` в список `List`.

Но мы можем сделать то же самое элегантнее, используя метод `Optional.stream()`, следующим образом:

```
// Предпочтительно
public List<Book> fetchBooksPrefer(List<String> isbnns) {
    return isbnns.stream()
        .map(this::fetchBookByIsbn)
        .flatMap(Optional::stream)
        .collect(toList());
}
```



Практически, в случаях подобного рода мы можем использовать метод `Optional.stream()` для замены методов `filter()` и `map()` на метод `flatMap()`.

Вызов метода `Optional.stream()` для каждого `Optional<Book>`, возвращаемого методом `fetchBookByIsbn()`, приведет к потоку `Stream<Book>`, содержащему один объект класса `Book` или ничего (пустому потоку). Если `Optional<Book>` не содержит значения (является пустым), то `Stream<Book>` также будет пустым. Использование метода `flatMap()` вместо метода `map()` позволит избежать результата с типом `Stream<Stream<Book>>`.

В качестве бонуса мы можем преобразовать объект `Optional` в список:

```
public static<T> List<T> optionalToList(Optional<T> optional) {
    return optional.stream().collect(toList());
}
```

248. Класс *Optional* и операции, чувствительные к идентичности

Операции, чувствительные к идентичности, включают эквивалентность ссылок (`==`), идентичность на основе хеша или синхронизацию.

Класс `Optional` является классом на основе значений, таким как `LocalDateTime`, поэтому следует избегать операций, чувствительных к идентичности.

Например, давайте проверим эквивалентность двух объектов класса `Optional` посредством оператора `==`:

```
Book book = new Book();
Optional<Book> op1 = Optional.of(book);
Optional<Book> op2 = Optional.of(book);

// Нежелательно
// op1 == op2 => false, expected true
if (op1 == op2) {
    System.out.println("op1 эквивалентен op2, (посредством ==)");
} else {
    System.out.println("op1 не эквивалентен op2, (посредством ==)");
}
```


Это даст следующий результат:

```
op1 не эквивалентен op2, (посредством ==)
```

Поскольку `op1` и `op2` не являются ссылками на один и тот же объект, они не эквивалентны, поэтому не согласуются с имплементацией оператора `==`.

Для того чтобы сравнить значения, нам нужно опереться на метод `equals()`:

```
// Предпочтительно
if (op1.equals(op2)) {
    System.out.println("op1 эквивалентен op2, (посредством equals())");
} else {
    System.out.println("op1 не эквивалентен op2, (посредством equals())");
}
```

Это даст следующий результат:

```
op1 эквивалентен op2, (посредством equals())
```

В контексте операций, чувствительных к идентичности, никогда не делайте ничего подобного (помните, что класс `Optional` основан на значении, и такие классы не должны использоваться для запирания на замок — подробнее см. <https://rules.sonarsource.com/java/tag/java8/RSPEC-3436>):

```
Optional<Book> book = Optional.of(new Book());
synchronized(book) {
    ...
}
```

249. Возвращение значения типа *boolean* при пустом объекте класса *Optional*

Допустим, что у нас есть следующий простой метод:

```
public static Optional<Cart> fetchCart(long userId) {
    // корзина для покупок заданного пользователя "userId" может быть null
    Cart cart = ...;

    return Optional.ofNullable(cart);
}
```

Теперь мы хотим написать метод с именем `cartIsEmpty()`, который вызывает метод `fetchCart()` и возвращает флаг `true`, если полученная корзина является пустой. До JDK 11 мы могли бы имплементировать этот метод на основе метода `Optional.isPresent()` следующим образом:

```
// Нежелательно (после JDK 11)
public static boolean cartIsEmpty(long id) {
    Optional<Cart> cart = fetchCart(id);

    return !cart.isPresent();
}
```

Это решение прекрасно работает, но оно не очень выразительно. Мы выполняем проверку на пустоту через присутствие, и нам приходится отрицать результат метода `isPresent()`.

Начиная с JDK 11, класс `Optional` дополнен новым методом `isEmpty()`. Как следует из его названия, это флаговый метод, который возвращает `true`, если тестируемый объект класса `Optional` является пустым. Таким образом, мы можем увеличить выразительность нашего решения следующим образом:

```
// Предпочтительно (после JDK 11)
public static boolean cartIsEmpty(long id) {
    Optional<Cart> cart = fetchCart(id);

    return cart.isEmpty();
}
```

Резюме

Готово! Последняя задача в главе рассмотрена. К настоящему моменту у вас должны быть все аргументы, необходимые для правильного использования класса `Optional`.

Вы можете скачать приложения из этой главы, чтобы увидеть результаты и получить дополнительные детали, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

13

API HTTP-клиента и протокола WebSocket

Эта глава содержит 20 задач, целью которых — охватить API HTTP-клиента и протокола WebSocket.

Помните, был такой подкласс `URLConnection`? Так вот, JDK 11 поставляется с API HTTP-клиента как переосмыслением подкласса `URLConnection`. API HTTP-клиента является простым в использовании и поддерживает протоколы HTTP/2 (по умолчанию) и HTTP/1.1. Для обеспечения обратной совместимости API HTTP-клиента автоматически переходит с HTTP/2 на HTTP/1.1, если сервер не поддерживает HTTP/2. Более того, API HTTP-клиента поддерживает синхронные и асинхронные модели программирования и опирается на потоки для передачи данных (реактивные потоки). Он также поддерживает протокол WebSocket, который используется в веб-приложениях реального времени с целью обеспечения связи между клиентом и сервером с низкими накладными расходами на пересылку сообщений.

Задачи

Используйте следующие задачи для проверки вашего умения программировать API HTTP-клиента и протокола WebSocket. Перед тем как обратиться к решениям и скачать примеры программ, настоятельно рекомендуется изучить каждую задачу.

250. **Протокол HTTP/2.** Представить краткий обзор протокола HTTP/2.
251. **Инициирование асинхронного запроса `GET`.** Написать программу, которая использует API HTTP-клиента с целью инициирования асинхронного запроса `GET` и вывода на экран кода и тела отклика.
252. **Настройка прокси-сервера.** Написать программу, которая использует API HTTP-клиента с целью настройки соединения через прокси-сервер.

- 253. **Настройка/получение заголовков.** Написать программу, которая добавляет добавочные заголовки к запросу и получает заголовки отклика.
- 254. **Настройка метода HTTP.** Написать программу, которая задает метод HTTP-запроса (например, GET, POST, PUT и DELETE).
- 255. **Настройка тела запроса.** Написать программу, которая использует API HTTP-клиента с целью добавления тела к запросу.
- 256. **Настройка аутентификации соединения.** Написать программу, которая использует API HTTP-клиента с целью настройки аутентификации соединения посредством пользовательского имени и пароля.
- 257. **Настройка таймаута.** Написать программу, которая использует API HTTP-клиента с целью задания времени (таймаута), в течение которого мы хотим ждать отклика.
- 258. **Настройка политики перенаправления.** Написать программу, которая использует API HTTP-клиента с целью автоматического перенаправления, если это необходимо.
- 258. **Отправка синхронных и асинхронных запросов.** Написать программу, которая отправляет один и тот же запрос в синхронном и асинхронном режимах.
- 260. **Обработка файлов cookie.** Написать программу, которая использует API HTTP-клиента с целью настройки обработчика файлов cookie.
- 261. **Получение информации об отклике.** Написать программу, которая использует API HTTP-клиента с целью получения информации об отклике (например, URI, версия, заголовки, код состояния, тело и т. д.).
- 262. **Обработка типов тел отклика.** Написать несколько фрагментов кода в целях иллюстрации того, как обрабатывать часто встречающиеся типы тел отклика посредством класса `HttpResponse.BodyHandlers`.
- 263. **Получение, обновление и сохранение данных JSON.** Написать программу, которая использует API HTTP-клиента с целью получения, обновления и сохранения данных JSON.
- 264. **Сжатие.** Написать программу, которая обрабатывает сжатые отклики (например, .gzip).
- 265. **Обработка данных формы.** Написать программу, которая использует API HTTP-клиента с целью отправки данных формы (`application/x-www-form-urlencoded`).
- 266. **Скачивание ресурса с сервера.** Написать программу, которая использует API HTTP-клиента с целью скачивания ресурса с сервера.
- 267. **Закачивание ресурса на сервер многочастным файлом.** Написать программу, которая использует API HTTP-клиента с целью закачивания ресурса на сервер.

268. **Серверная push-отправка по протоколу HTTP/2.** Написать программу, которая иллюстрирует средство HTTP/2 по push-отправке данных по инициативе сервера с помощью API HTTP-клиента.
269. **Протокол WebSocket.** Написать программу, которая открывает соединение с веб-сокетной конечной точкой, собирает данные в течение 10 секунд и закрывает соединение.

Решения

В следующих разделах описаны решения перечисленных выше задач. Помните, что обычно не существует одного единственно правильного варианта решения той или иной задачи. Кроме того, имейте в виду, что приведенные здесь объяснения включают только самые интересные и важные детали, необходимые для решения. Вы можете скачать примеры решений, чтобы увидеть дополнительные детали и поэкспериментировать с программами, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

250. Протокол HTTP/2

HTTP/2 — это эффективный протокол, который существенно и измеримо улучшает протокол HTTP/1.1.

В рамках общей картины HTTP/2 состоит из двух частей:

- ♦ **слой кадрирования** — это ключевая способность мультиплексирования протокола HTTP/2;
- ♦ **слой данных**, который содержит данные (то, что мы в типичной ситуации называем HTTP).

Схема на рис. 13.1 демонстрирует связь по протоколу HTTP/1.1 (вверху) и по протоколу HTTP/2 (внизу).



Рис. 13.1

Протокол HTTP/2 широко используется серверами и браузерами, и он идет в комплекте со следующими улучшениями по сравнению с протоколом HTTP/1.1.

- ◆ **Двоичный протокол.** Менее читаемый людьми, но более дружелюбный к машине, *слой кадрирования* HTTP/2 — это двоичный протокол.
- ◆ **Мультиплексирование.** Это относится к переплетенным запросам и откликам. Несколько запросов выполняются одновременно на одном и том же соединении.
- ◆ **Отправка данных по инициативе сервера** (серверная push-отправка). Сервер может решить отправить дополнительные ресурсы клиенту.
- ◆ **Единственное подключение к серверу.** HTTP/2 использует одну-единственную линию связи (ТСР-соединение) в расчете на источник (домен).
- ◆ **Сжатие заголовков.** HTTP/2 использует сжатие HPACK для уменьшения размера заголовков. Это оказывает значительное влияние на избыточные байты.
- ◆ **Шифрование.** Большая часть данных, передаваемых по проводам, зашифрована.

251. Инициирование асинхронного запроса *GET*

Инициирование асинхронного запроса GET — это трехшаговое задание, которое выполняется следующим образом:

1. Создать новый объект `HttpClient` (`java.net.http.HttpClient`):
2. Построить объект `HttpRequest` (`java.net.http.HttpRequest`) и указать запрос (по умолчанию это запрос GET):

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://reqres.in/api/users/2"))
    .build();
```



Для установки URI-идентификатора мы можем вызвать конструктор `HttpRequest.newBuilder(URI)` либо метод `uri(URI)` на экземпляре класса `Builder` (как мы сделали ранее).

3. Инициировать запрос и ждать отклика (`java.net.http.HttpResponse`). Будучи синхронным запросом, приложение блокирует работу до тех пор, пока ответ не станет доступен:

```
HttpResponse<String> response
    = client.send(request, BodyHandlers.ofString());
```

Если мы сгруппируем эти три шага и добавим строки кода для показа кода отклика и тела отклика на консоли, то получим следующий исходный код:

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://reqres.in/api/users/2"))
    .build();
```

```
HttpResponse<String> response
    = client.send(request, BodyHandlers.ofString());

System.out.println("Код статуса: " + response.statusCode());
System.out.println("\n Тело: " + response.body());
```

Один из возможных результатов для приведенного выше исходного кода выглядит так:

```
Код статуса: 200
Тело:
{
  "data": {
    "id": 2,
    "email": "janet.weaver@reqres.in",
    "first_name": "Janet",
    "last_name": "Weaver",
    "avatar": "https://s3.amazonaws.com/..."
  }
}
```

По умолчанию этот запрос осуществляется по протоколу HTTP/2. Однако мы также можем явно задать версию протокола посредством метода `HttpRequest.Builder.version()`. Этот метод получает аргумент типа `HttpClient.Version`, являющийся типом данных `enum`, который предоставляет две константы: `HTTP_2` и `HTTP_1_1`. Ниже приведен пример явного понижения до протокола HTTP/1.1:

```
HttpRequest request = HttpRequest.newBuilder()
    .version(HttpClient.Version.HTTP_1_1)
    .uri(URI.create("https://reqres.in/api/users/2"))
    .build();
```

Стандартные настройки для типа `HttpClient` следующие:

- ◆ HTTP/2;
- ◆ нет аутентификатора;
- ◆ нет таймаута соединения;
- ◆ нет обработчика файлов `cookie`;
- ◆ исполнитель по умолчанию пула нитей;
- ◆ политика перенаправления `NEVER`;
- ◆ прокси-селектор по умолчанию;
- ◆ контекст SSL по умолчанию.

В следующем разделе мы рассмотрим построителя параметров запроса.

Построитель параметров запроса

Работа с URI-идентификаторами, содержащими параметры запроса, предусматривает кодирование этих параметров. Встроенный в Java метод выполнения этой операции называется `URLEncoder.encode()`. Но конкатенирование и кодирование нескольких параметров запроса приводит к чему-то похожему на следующее:

```
URI uri = URI.create("http://localhost:8080/books?name=" +
    URLEncoder.encode("Games & Fun!", StandardCharsets.UTF_8) +
    "&no=" + URLEncoder.encode("124#442#000", StandardCharsets.UTF_8) +
    "&price=" + URLEncoder.encode("$23.99", StandardCharsets.UTF_8)
);
```

Когда приходится работать со значительным числом параметров запроса, такое решение не очень удобно. Однако мы можем попытаться написать вспомогательный метод, чтобы скрыть метод `URLEncoder.encode()` в цикле обхода коллекции параметров запроса, либо можем опереться на построителя URI-идентификатора.

В прикладном каркасе Spring построитель URI-идентификатора называется `org.springframework.web.util.UriComponentsBuilder`. Следующий код говорит сам за себя:

```
URI uri = UriComponentsBuilder.newInstance()
    .scheme("http")
    .host("localhost")
    .port(8080)
    .path("books")
    .queryParams("name", "Games & Fun!")
    .queryParams("no", "124#442#000")
    .queryParams("price", "$23.99")
    .build()
    .toUri();
```

В приложении, не использующем прикладной каркас Spring, мы можем опереться на построителя URI, такого как библиотека `urlbuilder` (<https://github.com/mikaelhg/urlbuilder>). Исходный код, прилагаемый к этой книге, содержит пример его использования.

252. Настройка прокси-селектора

Для настройки прокси-селектора (т. е. опосредующего селектора) мы опираемся на метод `HttpClient.proxy()` класса `Builder`. Метод `proxy()` получает аргумент типа `ProxySelector`, который может быть общесистемным прокси-селектором (посредством метода `getDefault()`) либо прокси-селектором, на который указывает его адрес (посредством `InetSocketAddress`).

Допустим, что у нас есть прокси по адресу `proxy.host:80`. Мы можем настроить этот прокси следующим образом:

```
HttpClient client = HttpClient.newBuilder()
    .proxy(ProxySelector.of(new InetSocketAddress("proxy.host", 80)))
    .build();
```

В качестве альтернативы мы можем настроить общесистемный прокси-селектор:

```
HttpClient client = HttpClient.newBuilder()
    .proxy(ProxySelector.getDefault())
    .build();
```

253. Настройка/получение заголовков

Классы `HttpRequest` и `HttpResponse` предоставляют набор методов для работы с заголовками. Мы узнаем об этих методах в следующих разделах.

Настройка заголовков запросов

Класс `HttpRequest.Builder` использует три метода для установки дополнительных заголовков:

◆ `header(String name, String value)` и `setHeader(String name, String value)` применяются для добавления заголовков по одному:

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(...)
    ...
    .header("key_1", "value_1")
    .header("key_2", "value_2")
    ...
    .build();
```

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(...)
    ...
    .setHeader("key_1", "value_1")
    .setHeader("key_2", "value_2")
    ...
    .build();
```



TIP

Разница между методами `header()` и `setHeader()` заключается в том, что первый добавляет указанный заголовок, а второй устанавливает указанный заголовок. Другими словами, метод `header()` добавляет заданное значение в список значений для этого имени/ключа, а метод `setHeader()` перезаписывает все ранее заданные значения для этого имени/ключа.

- ◆ `headers(String... headers)` используется для добавления заголовков, разделенных запятой:

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(...)
    ...
    .headers("key_1", "value_1", "key_2",
        "value_2", "key_3", "value_3", ...)
    ...
    .build();
```

Например, заголовки `Content-Type: application/json` и `Referer: https://reqres.in/` могут быть добавлены к запросу, который инициируется URI-идентификатором `https://reqres.in/api/users/2` следующим образом:

```
HttpRequest request = HttpRequest.newBuilder()
    .header("Content-Type", "application/json")
    .header("Referer", "https://reqres.in/")
    .uri(URI.create("https://reqres.in/api/users/2"))
    .build();
```

Вы также можете сделать следующее:

```
HttpRequest request = HttpRequest.newBuilder()
    .setHeader("Content-Type", "application/json")
    .setHeader("Referer", "https://reqres.in/")
    .uri(URI.create("https://reqres.in/api/users/2"))
    .build();
```

Наконец, вы можете сделать что-то вроде этого:

```
HttpRequest request = HttpRequest.newBuilder()
    .headers("Content-Type", "application/json", "Referer", "https://reqres.in/")
    .uri(URI.create("https://reqres.in/api/users/2"))
    .build();
```

В зависимости от цели, все три метода могут использоваться в сочетании для указания заголовков запроса.

Получение заголовков запроса/отклика

Получить заголовки запроса можно посредством метода `HttpRequest.headers()`. Аналогичный метод существует в классе `HttpResponse` для получения заголовков отклика. Оба метода возвращают объект класса `HttpHeaders`.

Оба эти метода могут использоваться одинаково, поэтому давайте сосредоточимся на получении заголовков отклика. И сделать это мы можем вот так:

```
HttpResponse<...> response ...
HttpHeaders allHeaders = response.headers();
```

Получить все значения заголовка можно посредством метода `HttpHeaders.allValues()` следующим образом:

```
List<String> allValuesOfCacheControl
    = response.headers().allValues("Cache-Control");
```

Получить только первое значение заголовка можно с помощью `HttpHeaders.firstValue()`:

```
Optional<String> firstValueOfCacheControl
    = response.headers().firstValue("Cache-Control");
```



Если возвращаемое значение заголовка имеет тип `Long`, то следует опираться на метод `HttpHeaders.firstValueAsLong()`. Этот метод получает аргумент, представляющий имя заголовка, и возвращает объект `Optional<Long>`. Если значение указанного заголовка не получается разобрать как тип `Long`, то будет выброшено исключение `NumberFormatException`.

254. Настройка метода HTTP

Мы можем указать метод HTTP, используемый нашим запросом, с помощью четырех следующих методов из класса `HttpRequest.Builder`.

◆ **GET()** отправляет запрос с помощью метода GET HTTP-протокола:

```
HttpRequest requestGet = HttpRequest.newBuilder()
    .GET() // can be omitted since it is default
    .uri(URI.create("https://reqres.in/api/users/2"))
    .build();
```

◆ **POST()** отправляет запрос с помощью метода POST HTTP-протокола:

```
HttpRequest requestPost = HttpRequest.newBuilder()
    .header("Content-Type", "application/json")
    .POST(HttpRequest.BodyPublishers.ofString(
        "{\"name\": \"morpheus\", \"job\": \"leader\"}"))
    .uri(URI.create("https://reqres.in/api/users"))
    .build();
```

◆ **PUT()** отправляет запрос с помощью метода PUT HTTP-протокола:

```
HttpRequest requestPut = HttpRequest.newBuilder()
    .header("Content-Type", "application/json")
    .PUT(HttpRequest.BodyPublishers.ofString(
        "{\"name\": \"morpheus\", \"job\": \"zion resident\"}"))
    .uri(URI.create("https://reqres.in/api/users/2"))
    .build();
```

◆ **DELETE()** отправляет запрос с помощью метода DELETE HTTP-протокола:

```
HttpRequest requestDelete = HttpRequest.newBuilder()
    .DELETE()
    .uri(URI.create("https://reqres.in/api/users/2"))
    .build();
```

Клиент может обрабатывать все типы методов HTTP-протокола, а не только предопределенные методы (GET, POST, PUT и DELETE). Для того чтобы создать запрос с другим методом HTTP-протокола, нам просто нужно вызвать метод `method()`.

Следующее решение инициирует запрос PATCH HTTP-протокола:

```
HttpRequest requestPatch = HttpRequest.newBuilder()
    .header("Content-Type", "application/json")
    .method("PATCH", HttpRequest.BodyPublishers.ofString(
        "{\"name\": \"morpheus\", \"job\": \"zion resident\"}"))
    .uri(URI.create("https://reqres.in/api/users/1"))
    .build();
```

Когда тело запроса не требуется, мы можем опереться на метод `BodyPublishers.noBody()`. Следующий вариант решения использует метод `noBody()` для инициирования запроса HEAD HTTP-протокола:

```
HttpRequest requestHead = HttpRequest.newBuilder()
    .method("HEAD", HttpRequest.BodyPublishers.noBody())
    .uri(URI.create("https://reqres.in/api/users/1"))
    .build();
```

В случае нескольких подобных запросов мы можем опереться на метод `copy()` для копирования строителя, как показано в следующем фрагменте кода:

```
HttpRequest.Builder builder = HttpRequest.newBuilder()
    .uri(URI.create("..."));

HttpRequest request1 = builder.copy().setHeader("...", "...").build();
HttpRequest request2 = builder.copy().setHeader("...", "...").build();
```

255. Настройка тела запроса

Настроить тело запроса можно с помощью методов `HttpRequest.Builder.POST()` и `HttpRequest.Builder.PUT()` или посредством метода `method()` (например, `method("PATCH", HttpRequest.BodyPublisher)`). `POST()` и `PUT()` берут аргумент типа `HttpRequest.BodyPublisher`. API идет в комплекте с несколькими имплементациями этого интерфейса (`BodyPublisher`) в классе `HttpRequest.BodyPublishers`, как показано ниже:

- ◆ `BodyPublishers.ofString();`
- ◆ `BodyPublishers.ofFile();`
- ◆ `BodyPublishers.ofByteArray();`
- ◆ `BodyPublishers.ofInputStream();`

Мы рассмотрим эти имплементации в следующих разделах.

Создание тела из строки текста

Создать тело из строки текста можно с помощью `BodyPublishers.ofString()`:

```
HttpRequest requestBody = HttpRequest.newBuilder()
    .header("Content-Type", "application/json")
    .POST(HttpRequest.BodyPublishers.ofString(
```

```
"{"name\: \"morpheus\", \"job\: \"leader\"}"))  
.uri (URI.create("https://reqres.in/api/users"))  
.build();
```

Для указания набора символов следует вызвать `ofString(String s, Charset charset)`.

Создание тела из потока *InputStream*

Создать тело из потока `InputStream` можно с помощью метода `BodyPublishers.ofInputStream()`, как показано в следующем ниже фрагменте кода (здесь мы опираемся на `ByteArrayInputStream`, но, разумеется, подойдет любой другой `InputStream`):

```
HttpRequest requestBodyOfInputStream = HttpRequest.newBuilder()  
.header("Content-Type", "application/json")  
.POST(HttpRequest.BodyPublishers.ofInputStream(()  
-> inputStream("user.json")))  
.uri (URI.create("https://reqres.in/api/users"))  
.build();  
  
private static ByteArrayInputStream inputStream(String fileName) {  
    try (ByteArrayInputStream inputStream = new ByteArrayInputStream(  
        Files.readAllBytes(Path.of(fileName)))) {  
  
        return inputStream;  
    } catch (IOException ex) {  
        throw new RuntimeException("Не получается прочитать файл", ex);  
    }  
}
```

Для того чтобы воспользоваться преимуществами ленивого создания, поток `InputStream` должен быть передан как поставщик (`Supplier`).

Создание тела из массива байтов

Создать тело из массива байтов можно с помощью метода `BodyPublishers.ofByteArray()`, как показано в следующем фрагменте кода:

```
HttpRequest requestBodyOfByteArray = HttpRequest.newBuilder()  
.header("Content-Type", "application/json")  
.POST(HttpRequest.BodyPublishers.ofByteArray(  
    Files.readAllBytes(Path.of("user.json"))))  
.uri (URI.create("https://reqres.in/api/users"))  
.build();
```

Мы также можем отправить только часть массива байтов, используя метод `ofByteArray(byte[] buf, int offset, int length)`. Кроме того, мы можем предоставить данные из `Iterable` байтовых массивов, используя метод `ofByteArrays(Iterable<byte[]> iter)`.

Создание тела из файла

Создать тело из файла можно с помощью метода `BodyPublishers.ofFile()`:

```
HttpRequest requestBodyOfFile = HttpRequest.newBuilder()
    .header("Content-Type", "application/json")
    .POST(HttpRequest.BodyPublishers.ofFile(Path.of("user.json")))
    .uri(URI.create("https://reqres.in/api/users"))
    .build();
```

256. Настройка аутентификации соединения

В типичной ситуации аутентификация на сервере выполняется посредством пользовательского имени и пароля. В кодовой форме это можно сделать с помощью класса `Authenticator` (он согласовывает учетные данные для HTTP-аутентификации) и класса `PasswordAuthentication` (держателя пользовательского имени и пароля) вместе, как показано ниже:

```
HttpClient client = HttpClient.newBuilder()
    .authenticator(new Authenticator() {

        @Override
        protected PasswordAuthentication getPasswordAuthentication() {

            return new PasswordAuthentication(
                "username",
                "password".toCharArray());
        }
    })
    .build();
```

Далее клиент может быть использован для отправки запросов:

```
HttpRequest request = HttpRequest.newBuilder()
    ...
    .build();

HttpResponse<String> response
    = client.send(request, HttpResponse.BodyHandlers.ofString());
```



Класс `Authenticator` поддерживает разные схемы аутентификации (например, базовую или дайджест-аутентификацию).

Еще одно решение состоит в добавлении учетных данных в заголовок:

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
    .header("Authorization", basicAuth("username", "password"))
```

```
...
    .build();

HttpResponse<String> response
    = client.send(request, HttpResponse.BodyHandlers.ofString());

private static String basicAuth(String username, String password) {
    return "Basic " + Base64.getEncoder().encodeToString(
        (username + ":" + password).getBytes());
}
```

В случае аутентификации носителя (токена HTTP-носителя) мы делаем следующее:

```
HttpRequest request = HttpRequest.newBuilder()
    .header("Authorization", "Bearer mT8JNMyWCG0D7waCHkyxo0Hm80YBqelv5SBL")
    .uri(URI.create("https://gorest.co.in/public-api/users"))
    .build();
```

Мы также можем сделать это в теле запроса POST:

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
    .header("Content-Type", "application/json")
    .POST(BodyPublishers.ofString("{\"email\":\"eve.holt@reqres.in\",
    \"password\":\"cityslicka\"}"))
    .uri(URI.create("https://reqres.in/api/login"))
    .build();
```

```
HttpResponse<String> response
    = client.send(request, HttpResponse.BodyHandlers.ofString());
```



Различные запросы могут использовать разные учетные данные. Кроме того, класс `Authenticator` предусматривает набор методов (например, метод `getRequestingSite()`), которые полезны, если мы хотим выяснить, какие значения должны быть предоставлены. В рабочей среде приложение не должно предоставлять учетные данные в открытом виде, как это было в приведенных выше примерах.

257. Настройка таймаута

По умолчанию запрос не имеет таймаута (имеет бесконечный таймаут). Для того чтобы установить количество времени (таймаут), в течение которого мы хотим дожидаться отклика, мы можем вызвать метод `HttpRequest.Builder.timeout()`. Он получает аргумент типа `Duration`, который можно использовать следующим образом:

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://reqres.in/api/users/2"))
    .timeout(Duration.of(5, ChronoUnit.MILLIS))
    .build();
```

Если указанный таймаут истекает, то будет выброшено исключение `java.net.http.HttpConnectTimeoutException`.

258. Настройка политики перенаправления

Когда мы пытаемся получить доступ к ресурсу, который был перемещен в другой URI, сервер возвращает код состояния HTTP в диапазоне 3xx, а также информацию о новом URI. Браузеры способны автоматически отправлять другой запрос в новое место, когда получают отклик на перенаправление (301, 302, 303, 307 и 308).

API HTTP-клиента может автоматически перенаправлять на этот новый URI, если мы явно зададим политику перенаправления посредством метода `followRedirects()`:

```
HttpClient client = HttpClient.newBuilder()
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build();
```

Для того чтобы никогда не выполнять перенаправление, следует просто передать константу `HttpClient.Redirect.NEVER` методу `followRedirects()` (это значение по умолчанию).

Для того чтобы всегда перенаправлять, за исключением URL-адресов HTTPS на URL-адреса HTTP, методу `followRedirects()` следует просто передать константу `HttpClient.Redirect.NORMAL`.

Если политика перенаправления не установлена равной `ALWAYS`, то за обработку перенаправлений отвечает приложение. Обычно это достигается путем чтения нового адреса из HTTP-заголовка `Location` следующим образом (приведенный далее код заинтересован в перенаправлении только в том случае, если возвращаемый код статуса равен 301 (перемещен постоянно) или 308 (постоянное перенаправление)):

```
int sc = response.statusCode();

if (sc == 301 || sc == 308) { // использовать enum для кодов HTTP-отклика
    String newLocation = response.headers()
        .firstValue("Location").orElse("");

    // обработать перенаправление на newLocation
}
```

Перенаправление можно легко обнаружить путем сравнения URI запроса с URI отклика. Если они не совпадают, то происходит перенаправление:

```
if (!request.uri().equals(response.uri())) {
    System.out.println("Запрос был перенаправлен: " + response.uri());
}
```


259. Отправка синхронных и асинхронных запросов

Отправить запрос на сервер можно с помощью следующих двух методов из класса `HttpClient`:

- ♦ метод `send()` отправляет запрос синхронно (он будет блокировать работу до тех пор, пока отклик не станет доступен либо не истечет таймаут);
- ♦ метод `sendAsync()` отправляет запрос асинхронно (не блокируя).

Мы объясним разные способы отправки запроса в следующих разделах.

Отправка запроса синхронно

Мы уже делали это в предыдущих задачах, и поэтому быстро напомним:

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://reqres.in/api/users/2"))
    .build();

HttpResponse<String> response
    = client.send(request, HttpResponse.BodyHandlers.ofString());
```

Отправка запроса асинхронно

Для асинхронной отправки запросов API HTTP-клиента опирается на класс `CompletableFuture`, как описано в *главе 11*, и метод `sendAsync()`, как показано ниже:

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://reqres.in/api/users/2"))
    .build();

client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .exceptionally(e -> "Exception: " + e)
    .thenAccept(System.out::println)
    .get(30, TimeUnit.SECONDS); // or join()
```

В качестве альтернативы допустим, что, ожидая отклика, мы хотим выполнять и другие операции:

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://reqres.in/api/users/2"))
    .build();
```

```

CompletableFuture<String> response
    = client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .exceptionally(e -> "Исключение: " + e);

while (!response.isDone()) {
    Thread.sleep(50);
    System.out.println("Выполнить другие операции во время ожидания отклика...");
}

String body = response.get(30, TimeUnit.SECONDS); // либо join()
System.out.println("Тело: " + body);

```

Одновременная отправка многочисленных запросов

Как отправить несколько запросов одновременно и дождаться доступности всех откликов?

Как мы знаем, класс `CompletableFuture` идет в комплекте с методом `allOf()` (для получения более подробной информации см. главу 11), который может исполнять задачи в параллельном режиме и ждать, когда они все завершатся. Он возвращает `CompletableFuture<Void>`.

Следующий ниже код ожидает отклика на четыре запроса:

```

List<URI> uris = Arrays.asList(
    new URI("https://reqres.in/api/users/2"), // один пользователь user
    new URI("https://reqres.in/api/users?page=2"), // список пользователей
    new URI("https://reqres.in/api/unknown/2"), // список ресурсов
    new URI("https://reqres.in/api/users/23")); // пользователь не найден

HttpClient client = HttpClient.newHttpClient();

List<HttpRequest> requests = uris.stream()
    .map(HttpRequest::newBuilder)
    .map(reqBuilder -> reqBuilder.build())
    .collect(Collectors.toList());

CompletableFuture.allOf(requests.stream()
    .map(req -> client.sendAsync(req, HttpResponse.BodyHandlers.ofString())
    .thenApply((res) -> res.uri() + " | " + res.body() + "\n")
    .exceptionally(e -> "Исключение: " + e)
    .thenAccept(System.out::println))
    .toArray(CompletableFuture<?>[]::new))
    .join();

```

Для того чтобы собрать тела откликов (например, в список `List<String>`), рассмотрим класс `WaitAllResponsesFetchBodiesInList`, который имеется в исходном коде, прилагаемом к этой книге.

Применить собственный объект `Executor` можно следующим образом:

```
ExecutorService executor = Executors.newFixedThreadPool(5);
```

```
HttpClient client = HttpClient.newBuilder()
    .executor(executor)
    .build();
```

260. Обработка файлов cookie

По умолчанию HTTP-клиент JDK 11 поддерживает файлы cookie, но есть случаи, когда встроенная поддержка отключена. Мы можем включить ее следующим образом:

```
HttpClient client = HttpClient.newBuilder()
    .cookieHandler(new CookieManager())
    .build();
```

Таким образом, API HTTP-клиента позволяет нам устанавливать обработчик файлов cookie с помощью метода `HttpClient.Builder.cookieHandler()`. Этот метод получает аргумент типа `CookieManager`.

Следующее ниже решение настраивает объект `CookieManager`, который не принимает файлы cookie:

```
HttpClient client = HttpClient.newBuilder()
    .cookieHandler(new CookieManager(null, CookiePolicy.ACCEPT_NONE))
    .build();
```

Для приема файлов cookies нужно задать политике `CookiePolicy` значение `ALL` (принимать все файлы cookie) либо `ACCEPT_ORIGINAL_SERVER` (принимать файлы cookie только с исходного сервера).

Приведенные далее варианты решения принимают все файлы cookie и показывают их в консоли (если какие-либо учетные данные считаются недействительными, то следует рассмотреть возможность получения нового токена из <https://gorest.co.in/rest-console.html>):

```
CookieManager cm = new CookieManager();
cm.setCookiePolicy(CookiePolicy.ACCEPT_ALL);
```

```
HttpClient client = HttpClient.newBuilder()
    .cookieHandler(cm)
    .build();
```

```
HttpRequest request = HttpRequest.newBuilder()
    .header("Authorization", "Bearer mT8JNMyWCGOD7waCHkyxo0Hm80YBqelv5SBL")
    .uri(URI.create("https://gorest.co.in/public-api/users/1"))
    .build();
```

```
HttpResponse<String> response
    = client.send(request, HttpResponse.BodyHandlers.ofString());

System.out.println("Код статуса: " + response.statusCode());
System.out.println("\n Тело: " + response.body());

CookieStore cookieStore = cm.getCookieStore();
System.out.println("\nФайлы cookie: " + cookieStore.getCookies());
```

Проверить заголовок set-cookie может следующим образом:

```
Optional<String> setcookie = response.headers().firstValue("set-cookie");
```

261. Получение информации отклика

В получении информации отклика мы можем опереться на методы из класса `HttpResponse`. Имена этих методов очень понятны, поэтому следующий ниже фрагмент кода не требует пояснений:

```
...
HttpResponse<String> response
    = client.send(request, HttpResponse.BodyHandlers.ofString());

System.out.println("Версия: " + response.version());
System.out.println("\nURI: " + response.uri());
System.out.println("\nКод статуса: " + response.statusCode());
System.out.println("\nЗаголовки: " + response.headers());
System.out.println("\nТело: " + response.body());
```

Изучите документацию, чтобы найти более полезные методы.

262. Обработка тела отклика

Обработать тело отклика можно с помощью интерфейса `HttpResponse.BodyHandler`. API идет в комплекте с несколькими имплементациями этого интерфейс (`BodyHandler`) в классе `HttpResponse.BodyHandlers`:

- ◆ `BodyHandlers.ofByteArray();`
- ◆ `BodyHandlers.ofFile();`
- ◆ `BodyHandlers.ofString();`
- ◆ `BodyHandlers.ofInputStream();`
- ◆ `BodyHandlers.ofLines();`

Рассматривая следующий ниже запрос, давайте взглянем на несколько решений для обработки тела отклика:

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://reqres.in/api/users/2"))
    .build();
```

В следующих разделах мы посмотрим, как обращаться с разными типами тел отклика.

Обработка тела отклика как строки

Обработать тело отклика как строку можно с помощью метода `BodyHandlers.ofString()`, как показано в следующем фрагменте кода:

```
HttpResponse<String> responseOfString
    = client.send(request, HttpResponse.BodyHandlers.ofString());

System.out.println("Код статуса: " + responseOfString.statusCode());
System.out.println("Тело: " + responseOfString.body());
```

Для указания набора символов нужно вызвать метод `ofString(String s, Charset charset)`.

Обработка тела отклика как файла

Обработать тело отклика как файл можно с помощью метода `BodyHandlers.ofFile()`:

```
HttpResponse<Path> responseOfFile = client.send(
    request, HttpResponse.BodyHandlers.ofFile(Path.of("response.json")));

System.out.println("Код статуса: " + responseOfFile.statusCode());
System.out.println("Тело: " + responseOfFile.body());
```

Для указания вариантов открытия следует вызвать метод `ofFile(Path file, OpenOption... openOptions)`.

Обработка тела отклика как массива байтов

Обработать тело отклика как массив байтов можно с помощью метода `BodyHandlers.ofByteArray()`:

```
HttpResponse<byte[]> responseOfByteArray = client.send(
    request, HttpResponse.BodyHandlers.ofByteArray());

System.out.println("Код статуса: " + responseOfByteArray.statusCode());
System.out.println("Тело: " + new String(responseOfByteArray.body()));
```

Для потребления массива байтов следует вызвать метод `ofByteArrayConsumer(Consumer<Optional<byte[]>> consumer)`.

Обработка тела отклика как входного потока

Обработать тело отклика как входной поток можно с помощью метода `BodyHandlers.ofInputStream()`:

```
HttpResponse<InputStream> responseOfInputStream = client.send(
    request, HttpResponse.BodyHandlers.ofInputStream());
```

```
System.out.println("\nHttpResponse.BodyHandlers.ofInputStream()");
System.out.println("Код статуса: " + responseOfInputStream.statusCode());

byte[] allBytes;

try (InputStream fromIs = responseOfInputStream.body()) {
    allBytes = fromIs.readAllBytes();
}

System.out.println("Тело: " + new String(allBytes, StandardCharsets.UTF_8));
```

Обработка тела отклика как потока строк

Обработать тело отклика как поток строк можно с помощью метода `BodyHandlers.ofLines()`:

```
HttpResponse<Stream<String>> responseOfLines = client.send(
    request, HttpResponse.BodyHandlers.ofLines());

System.out.println("Код статуса: " + responseOfLines.statusCode());
System.out.println("Тело: " + responseOfLines.body().collect(toList()));
```

263. Получение, обновление и сохранение данных JSON

В предыдущих задачах мы манипулировали данными JSON как открытым текстом (строками). API HTTP-клиента не предусматривает специальной или выделенной поддержки для данных JSON и обрабатывает этот тип данных как любую другую строку.

Тем не менее мы привыкли представлять данные JSON как обыкновенные объекты Java (POJO-объекты) и опираться на конвертирование между JSON и Java, когда это необходимо. Мы можем написать решение нашей задачи без привлечения API HTTP-клиента. Однако мы также можем написать решение, опираясь на собственную имплементацию интерфейса `HttpResponse.BodyHandler`, которая использует анализатор JSON для конвертирования отклика в объекты Java. Например, мы можем применить декларативную привязку JSON-B (представленную в главе 6).

Имплементирование интерфейса `HttpResponse.BodyHandler` подразумевает переопределение метода `apply(HttpResponse.ResponseInfo responseInfo)`. Используя этот метод, мы можем взять байты из отклика и конвертировать их в объект Java. Исходный код выглядит следующим образом:

```
public class JsonBodyHandler<T> implements HttpResponse.BodyHandler<T> {
    private final Jsonb jsonb;
    private final Class<T> type;

    private JsonBodyHandler(Jsonb jsonb, Class<T> type) {
        this.jsonb = jsonb;
    }
}
```

```
        this.type = type;
    }

    public static <T> JsonBodyHandler<T> jsonBodyHandler(Class<T> type) {
        return jsonBodyHandler(JsonbBuilder.create(), type);
    }

    public static <T> JsonBodyHandler<T> jsonBodyHandler(
        Jsonb jsonb, Class<T> type) {
        return new JsonBodyHandler<>(jsonb, type);
    }

    @Override
    public HttpResponse.BodySubscriber<T> apply(
        HttpResponse.ResponseInfo responseInfo) {

        return BodySubscribers.mapping(BodySubscribers.ofByteArray(),
            byteArray -> this.jsonb.fromJson(
                new ByteArrayInputStream(byteArray), this.type));
    }
}
```

Допустим, что данные JSON, которыми мы хотим управлять, выглядят следующим образом (это отклик, поступивший от сервера):

```
{
  "data": {
    "id": 2,
    "email": "janet.weaver@reqres.in",
    "first_name": "Janet",
    "last_name": "Weaver",
    "avatar": "https://s3.amazonaws.com/..."
  }
}
```

Объекты Java для представления этих данных JSON следующие:

```
public class User {
    private Data data;
    private String updatedAt;

    // геттеры, сеттеры и toString()
}

public class Data {
    private Integer id;
    private String email;

    @JsonbProperty("first_name")
    private String firstName;
```

```
@JsonbProperty("last_name")
private String lastName;

private String avatar;

// геттеры, сеттеры и toString()
}
```

Теперь давайте посмотрим, каким образом можно манипулировать данными JSON в запросах и откликах.

JSON-отклик в объект *User*

Следующее решение иницирует запрос GET и конвертирует возвращенный JSON-отклик в объект *User*:

```
Jsonb jsonb = JsonbBuilder.create();
HttpClient client = HttpClient.newHttpClient();

HttpRequest requestGet = HttpRequest.newBuilder()
    .uri(URI.create("https://reqres.in/api/users/2"))
    .build();

HttpResponse<User> responseGet = client.send(
    requestGet, JsonBodyHandler.jsonBodyHandler(jsonb, User.class));

User user = responseGet.body();
```

Обновленный объект *User* в JSON-запрос

Следующее ниже решение обновляет адрес электронной почты пользователя, которого мы выбрали в предыдущем подразделе:

```
user.getData().setEmail("newemail@gmail.com");

HttpRequest requestPut = HttpRequest.newBuilder()
    .header("Content-Type", "application/json")
    .uri(URI.create("https://reqres.in/api/users"))
    .PUT(HttpRequest.BodyPublishers.ofString(jsonb.toJson(user)))
    .build();

HttpResponse<User> responsePut = client.send(
    requestPut, JsonBodyHandler.jsonBodyHandler(jsonb, User.class));

User updatedUser = responsePut.body();
```


Новый объект *User* в JSON-запрос

Следующее ниже решение создает нового пользователя (статусный код отклика должен быть равен 201):

```
Data data = new Data();
data.setId(10);
data.setFirstName("John");
data.setLastName("Year");
data.setAvatar("https://johnyear.com/jy.png");

User newUser = new User();
newUser.setData(data);

HttpRequest requestPost = HttpRequest.newBuilder()
    .header("Content-Type", "application/json")
    .uri(URI.create("https://reqres.in/api/users"))
    .POST(HttpRequest.BodyPublishers.ofString(jsonb.toJson(user)))
    .build();

HttpResponse<Void> responsePost = client.send(
    requestPost, HttpResponse.BodyHandlers.discarding());

int sc = responsePost.statusCode(); // 201
```

Обратите внимание, что мы игнорируем любое тело отклика посредством метода `HttpResponse.BodyHandlers.discarding()`.

264. Сжатие

Обеспечение возможности сжатия `.gzip` на сервере — это часто встречающийся практический прием, который призван значительно улучшить время загрузки веб-сайта. Но API HTTP-клиента JDK 11 не использует преимущества сжатия `.gzip`. Другими словами, API HTTP-клиента не требует сжатых откликов и не знает, как обращаться с такими откликами.

Для того чтобы запросить сжатые отклики, мы должны отправить заголовок `Accept-Encoding` со значением `.gzip`. API HTTP-клиента не добавляет этот заголовок, поэтому мы добавим его сами следующим образом:

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
    .header("Accept-Encoding", "gzip")
    .uri(URI.create("https://davidwalsh.name"))
    .build();
```

Это только половина работы. На данный момент, если возможность кодировки `gzip` обеспечивается на сервере, мы будем получать сжатый отклик. Для того чтобы ус-

тановить, является ли отклик сжатым, мы должны проверить заголовок Encoding следующим образом:

```

HttpResponse<InputStream> response = client.send(
    request, HttpResponse.BodyHandlers.ofInputStream());

String encoding = response.headers()
    .firstValue("Content-Encoding").orElse("");

if ("gzip".equals(encoding)) {
    String gzipAsString = gZipToString(response.body());
    System.out.println(gzipAsString);
} else {
    String isAsString = isToString(response.body());
    System.out.println(isAsString);
}

```

Метод `gZipToString()` является вспомогательным методом, который берет поток `InputStream` и трактует его как поток `GZIPInputStream`. Другими словами, этот метод читает байты из заданного входного потока и использует их для создания строкового значения:

```

public static String gZipToString(InputStream gzip) throws IOException {
    byte[] allBytes;
    try (InputStream fromIs = new GZIPInputStream(gzip)) {
        allBytes = fromIs.readAllBytes();
    }
    return new String(allBytes, StandardCharsets.UTF_8);
}

```

Если отклик не сжат, то нам нужен вспомогательный метод `isToString()`:

```

public static String isToString(InputStream is) throws IOException {
    byte[] allBytes;
    try (InputStream fromIs = is) {
        allBytes = fromIs.readAllBytes();
    }
    return new String(allBytes, StandardCharsets.UTF_8);
}

```

265. Обработка данных формы

API HTTP-клиента в JDK 11 не имеет встроенной поддержки для инициирования POST-запросов в формате `x-www-form-urlencoded`. Решение этой проблемы опирается на собственный класс `BodyPublisher`.

Написать собственный класс `BodyPublisher` довольно просто, если мы рассмотрим следующее:

- ◆ данные представлены в виде пар "ключ–значение";
- ◆ каждая пара представлена в виде `ключ = значение`;

- ◆ пары разделяются символом &;
- ◆ ключи и значения надлежаще кодированы.

Поскольку данные представлены в виде пар "ключ–значение", их очень удобно хранить в коллекции Map. Далее мы просто обходим эту коллекцию в цикле и применяем приведенную выше информацию следующим образом:

```
public class FormBodyPublisher {
    public static HttpRequest.BodyPublisher ofForm(Map<Object, Object> data) {
        StringBuilder body = new StringBuilder();

        for (Object dataKey: data.keySet()) {
            if (body.length() > 0) {
                body.append("&");
            }

            body.append(encode(dataKey))
                .append("=")
                .append(encode(data.get(dataKey)));
        }

        return HttpRequest.BodyPublishers.ofString(body.toString());
    }

    private static String encode(Object obj) {
        return URLEncoder.encode(obj.toString(), StandardCharsets.UTF_8);
    }
}
```

Опираясь на это решение, запрос POST (x-www-form-urlencoded) может быть иницирован следующим образом:

```
Map<Object, Object> data = new HashMap<>();
data.put("firstname", "John");
data.put("lastname", "Year");
data.put("age", 54);
data.put("avatar", "https://avatars.com/johnyear");

HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
    .header("Content-Type", "application/x-www-form-urlencoded")
    .uri(URI.create("http://jorpela.fi/cgi-bin/echo.cgi"))
    .POST(FormBodyPublisher.ofForm(data))
    .build();

HttpResponse<String> response = client.send(
    request, HttpResponse.BodyHandlers.ofString());
```

В этом случае откликом является просто эхо отправленных данных. Приложение должно действовать в зависимости от отклика сервера, как показано в разд. "262. Обработка тела отклика" ранее в этой главе.

266. Скачивание ресурса с сервера

Как мы видели в разд. 255 "Настройка тела запроса" и 262 "Обработка тела отклика" ранее в этой главе, API HTTP-клиента может отправлять и получать текстовые и двоичные данные (например, изображения, видео и т. д.).

Скачивание ресурса с сервера зависит от следующих двух координат:

- ◆ отправка запроса GET;
- ◆ обработка полученных байтов (например, посредством метода `BodyHandlers.ofFile()`).

Следующий фрагмент кода скачивает файл `hibernate-core-5.4.2.Final.jar` из репозитория Maven для направления его к местоположению классов проекта:

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("http://.../hibernate-core-5.4.2.Final.jar"))
    .build();
```

```
HttpResponse<Path> response
    = client.send(request, HttpResponse.BodyHandlers.ofFile(
        Path.of("hibernate-core-5.4.2.Final.jar")));
```

Если скачиваемый ресурс поставляется посредством HTTP-заголовка `Content-Disposition`, т. е. имеет вид `Content-Disposition attachment;filename="..."`, то мы можем опереться на метод `BodyHandlers.ofFileDownload()`, как в следующем примере:

```
import static java.nio.file.StandardOpenOption.CREATE;
...
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("http://...downloadfile.php
        ?file=Hello.txt&cd=attachment+filename"))
    .build();
```

```
HttpResponse<Path> response = client.send(request,
    HttpResponse.BodyHandlers.ofFileDownload(Path.of(
        System.getProperty("user.dir")), CREATE));
```

Еще больше файлов, которые можно протестировать, доступно вот по этому адресу: http://demo.borland.com/testsite/download_testpage.php.

267. Закачивание ресурса на сервер многочастным файлом

Как мы видели в *разд. 255 "Настройка тела запроса"* ранее в этой главе, можно отправить файл (текстовый или двоичный) на сервер посредством метода `BodyPublishers.ofFile()` и POST-запроса.

Но отправка классического запроса с целью закачивания ресурса на сервер может предусматривать многочастную форму POST с заголовком `Content-Type` в формате `multipart/form-data`.

В этом случае тело запроса состоит из частей, разделенных границей, как показано на *рис. 13.2* (`--779d334bbfa...` является границей).

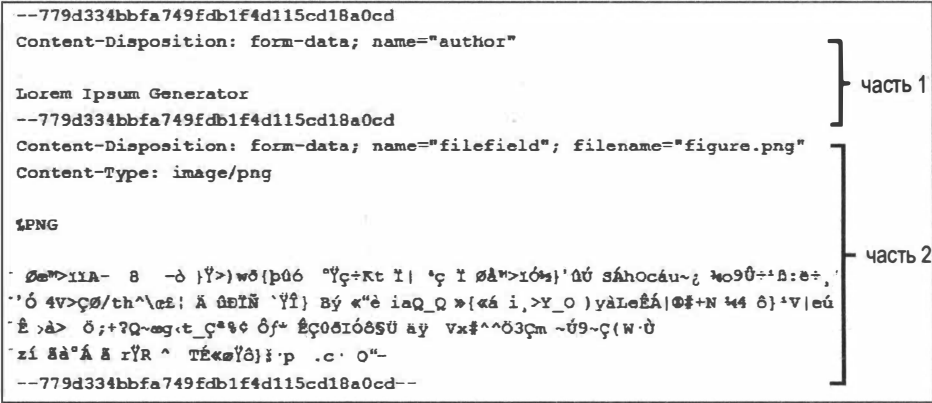


Рис. 13.2

Однако API HTTP-клиента JDK 11 не обеспечивает встроенную поддержку для построения тела запроса такого типа. Тем не менее, следуя приведенному снимку экрана на *рис. 13.2*, мы можем определить собственный класс `BodyPublisher`:

```
public class MultipartBodyPublisher {  
    private static final String LINE_SEPARATOR = System.lineSeparator();  
  
    public static HttpRequest.BodyPublisher ofMultipart(  
        Map<Object, Object> data, String boundary) throws IOException {  
  
        final byte[] separator = ("--" + boundary +  
            LINE_SEPARATOR + "Content-Disposition: form-data;  
            name = ").getBytes(StandardCharsets.UTF_8);  
  
        final List<byte[] > body = new ArrayList<>();  
  
        for (Object dataKey: data.keySet()) {  
            body.add(separator);  
            Object dataValue = data.get(dataKey);
```

```
if (dataValue instanceof Path) {
    Path path = (Path) dataValue;
    String mimeType = fetchMimeType(path);

    body.add(("\"" + dataKey + "\"; filename=\"\" +
    path.getFileName() + "\"" + LINE_SEPARATOR +
    "Content-Type: " + mimeType + LINE_SEPARATOR +
    LINE_SEPARATOR).getBytes(StandardCharsets.UTF_8));

    body.add(Files.readAllBytes(path));
    body.add(LINE_SEPARATOR.getBytes(StandardCharsets.UTF_8));
} else {
    body.add(("\"" + dataKey + "\"" + LINE_SEPARATOR +
    LINE_SEPARATOR + dataValue + LINE_SEPARATOR)
    .getBytes(StandardCharsets.UTF_8));
}
}

body.add(("--" + boundary + "--").getBytes(StandardCharsets.UTF_8));

return HttpRequest.BodyPublishers.ofByteArrays(body);
}

private static String fetchMimeType(
    Path filenamePath) throws IOException {

    String mimeType = Files.probeContentType(filenamePath);

    if (mimeType == null) {
        throw new IOException("Mime type could not be fetched");
    }

    return mimeType;
}
}
```

Теперь мы можем создать составной запрос следующим образом (мы пытаемся за-
качать текстовый файл под названием LoremIpsum.txt на сервер, который просто от-
правил обратно сырые данные формы):

```
Map<Object, Object> data = new LinkedHashMap<>();
data.put("author", "Lorem Ipsum Generator");
data.put("filefield", Path.of("LoremIpsum.txt"));

String boundary = UUID.randomUUID().toString().replaceAll("-", "");

HttpClient client = HttpClient.newHttpClient();
```

```
HttpRequest request = HttpRequest.newBuilder()
    .header("Content-Type", "multipart/form-data;boundary=" + boundary)
    .POST(MultipartBodyPublisher.ofMultipart(data, boundary))
    .uri(URI.create("http://jorpela.fi/cgi-bin/echoraw.cgi"))
    .build();
```

```
HttpResponse<String> response = client.send(
    request, HttpResponse.BodyHandlers.ofString());
```

Отклик должен быть похож на приведенный ниже (границей является просто случайный UUID):

```
--7ea7a8311ada4804ab11d29bcdedcc55
Content-Disposition: form-data; name="author"
Lorem Ipsum Generator
--7ea7a8311ada4804ab11d29bcdedcc55
Content-Disposition: form-data; name="filefield";
filename="LoremIpsum.txt"
Content-Type: text/plain
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua.
--7ea7a8311ada4804ab11d29bcdedcc55--
```

268. Серверная push-отправка по протоколу HTTP/2

Помимо *мультиплексирования*, еще одним мощным средством протокола HTTP/2 является способность отправлять данные по инициативе сервера (серверная push-отправка).

При традиционном подходе (HTTP/1.1) браузер запускает запрос на получение HTML-страницы и проводит разбор полученной разметки с целью идентификации ссылочных ресурсов (например, JS, CSS, изображений и т. д.). Для извлечения этих ресурсов браузер отправляет дополнительные запросы (по одному запросу на каждый ресурс, на который браузер ссылается). Протокол HTTP/2 также отправляет HTML-страницу и ссылочные ресурсы без явных запросов от браузера. Таким образом, браузер запрашивает HTML-страницу и получает страницу и все остальное, что необходимо для вывода страницы на экран.

API HTTP-клиента поддерживает это средство протокола HTTP/2 через интерфейс `PushPromiseHandler`. Имплементация этого интерфейса должна быть задана в качестве третьего аргумента метода `send()` или `sendAsync()`.

Интерфейс `PushPromiseHandler` опирается на три координаты, а именно:

- ◆ иницирующий клиент отправляет запрос (`initiatingRequest`);
- ◆ синтетический push-запрос (`pushPromiseRequest`);

- ◆ акцепторная функция, которая должна быть успешно активирована, чтобы принять обещание отправки, инициируемой сервером (акцептор).

Обещание отправки, инициируемой сервером, принимается путем вызова заданной акцепторной функции. Акцепторной функции должен быть передан непустой `BodyHandler`, который используется для обработки тела отклика в обещании. Акцепторная функция возвращает экземпляр `CompletableFuture`, который завершает отклик обещания.

Приняв во внимание вышеизложенное, рассмотрим имплементацию интерфейса `PushPromiseHandler`:

```
private static final List<CompletableFuture<Void>>
    asyncPushRequests = new CopyOnWriteArrayList<>();
...
private static HttpResponse.PushPromiseHandler<String>
    pushPromiseHandler() {
    return (HttpRequest initiatingRequest,
        HttpRequest pushPromiseRequest,
        Function<HttpResponse.BodyHandler<String> ,
        CompletableFuture<HttpResponse<String>>> acceptor) -> {
        CompletableFuture<Void> pushcf =
            acceptor.apply(HttpResponse.BodyHandlers.ofString())
                .thenApply(HttpResponse::body)
                .thenAccept((b) -> System.out.println("\nТело push-ресурса:\n " + b));

        asyncPushRequests.add(pushcf);

        System.out.println("\nТолько что получен номер push-обещания: " +
            asyncPushRequests.size());
        System.out.println("\nПервоначальный push-запрос: " +
            initiatingRequest.uri());
        System.out.println("Первоначальные push-заголовки: " +
            initiatingRequest.headers());
        System.out.println("Push-запрос обещания: " +
            pushPromiseRequest.uri());
        System.out.println("Push-заголовки обещания: " +
            pushPromiseRequest.headers());
    };
}
```

Теперь иницилируем запрос и передадим эту имплементацию `PushPromiseHandler` в метод `sendAsync()`:

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://http2.golang.org/serverpush"))
    .build();
```



```
client.sendAsync(request,
    HttpResponse.BodyHandlers.ofString(), pushPromiseHandler())
    .thenApply(HttpResponse::body)
    .thenAccept((b) -> System.out.println("\nГлавный ресурс:\n" + b))
    .join();

asyncPushRequests.forEach(CompletableFuture::join);
System.out.println("\nПолучено всего " +
    asyncPushRequests.size() + " push-запросов");
```

Если мы хотим вернуть обработчик push-запроса, который накапливает push-запросы и их отклики, в заданной коллекции Map, то мы можем опереться на метод `PushPromiseHandler.of()` следующим образом:

```
private static final ConcurrentMap<HttpRequest,
    CompletableFuture<HttpResponse<String>>> promisesMap
    = new ConcurrentHashMap<>();

private static final Function<HttpRequest,
    HttpResponse.BodyHandler<String>> promiseHandler
    = (HttpRequest req) -> HttpResponse.BodyHandlers.ofString();

public static void main(String[] args)
    throws IOException, InterruptedException {

    HttpClient client = HttpClient.newHttpClient();

    HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create("https://http2.golang.org/serverpush"))
        .build();

    client.sendAsync(request,
        HttpResponse.BodyHandlers.ofString(), pushPromiseHandler())
        .thenApply(HttpResponse::body)
        .thenAccept((b) -> System.out.println("\nГлавный ресурс:\n" + b))
        .join();

    System.out.println("\nРазмер коллекции Map с push-обещаниями: " +
        promisesMap.size() + "\n");

    promisesMap.entrySet().forEach((entry) -> {
        System.out.println("Request = " + entry.getKey() +
            ", \nОтклик = " + entry.getValue().join().body());
    });
}
```

```
private static HttpResponse.PushPromiseHandler<String>
    pushPromiseHandler() {
        return HttpResponse.PushPromiseHandler
            .of(promiseHandler, promisesMap);
    }
```

В обоих предыдущих вариантах решения мы использовали `BodyHandler` с типом `String` посредством метода `ofString()`. Такой подход не очень полезен, если сервер также отправляет двоичные данные (например, изображения). Поэтому, если мы имеем дело с двоичными данными, нам нужно переключиться на `BodyHandler` с типом `byte[]` посредством метода `ofByteArray()`. В качестве альтернативы мы можем отправить инициированные сервером ресурсы на диск посредством метода `ofFile()`, как показано в следующем варианте решения, который представляет собой адаптированную версию предыдущего варианта решения:

```
private static final ConcurrentMap<HttpRequest,
    CompletableFuture<HttpResponse<Path>>>
    promisesMap = new ConcurrentHashMap<>();

private static final Function<HttpRequest,
    HttpResponse.BodyHandler<Path>> promiseHandler
    = (HttpRequest req) -> HttpResponse.BodyHandlers.ofFile(
        Paths.get(req.uri().getPath()).getFileName());

public static void main(String[] args)
    throws IOException, InterruptedException {

    HttpClient client = HttpClient.newHttpClient();

    HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create("https://http2.golang.org/serverpush"))
        .build();

    client.sendAsync(request, HttpResponse.BodyHandlers.ofFile(
        Path.of("index.html")), pushPromiseHandler())
        .thenApply(HttpResponse::body)
        .thenAccept((b) -> System.out.println("\nГлавный ресурс:\n" + b))
        .join();

    System.out.println("\nРазмер коллекции Map с push-обещаниями: " +
        promisesMap.size() + "\n");

    promisesMap.entrySet().forEach((entry) -> {
        System.out.println("Запрос = " + entry.getKey() +
            ", \nResponse = " + entry.getValue().join().body());
    });
}
```

```
private static HttpResponseMessage.PushPromiseHandler<Path> pushPromiseHandler() {
    return HttpResponseMessage.PushPromiseHandler
        .of(promiseHandler, promisesMap);
}
```

Приведенный исходный код должен сохранить инициированные сервером ресурсы в пути к местоположению классов приложения (рис. 13.3).

	godocs	5/16/2019 10:26 AM	JScript Script File	18 KB
	index	5/16/2019 10:26 AM	Chrome HTML Do...	66 KB
	jquery.min	5/16/2019 10:26 AM	JScript Script File	92 KB
	playground	5/16/2019 10:26 AM	JScript Script File	15 KB
	style	5/16/2019 10:26 AM	Cascading Style S...	14 KB

Рис. 13.3

269. Протокол WebSocket

API HTTP-клиента поддерживает протокол WebSocket. С точки зрения API стержнем его имплементации является интерфейс `java.net.http.WebSocket`. Этот интерфейс выставляет наружу набор методов для обработки общения по протоколу WebSocket.

Построить экземпляр `WebSocket` асинхронно можно посредством метода `HttpClient.newWebSocketBuilder().buildAsync()`.

Например, мы можем подключиться к известной веб-сокетной конечной точке **Meetup RSVP** (<ws://stream.meetup.com/2/rsvps>) следующим образом:

```
HttpClient client = HttpClient.newHttpClient();
```

```
WebSocket webSocket = client.newWebSocketBuilder()
    .buildAsync(URI.create("ws://stream.meetup.com/2/rsvps"),
        wsListener).get(10, TimeUnit.SECONDS);
```

По своей природе протокол WebSocket является двунаправленным. В отправке данных мы можем опираться на методы `sendText()`, `sendBinary()`, `sendPing()` и `sendPong()`. Конечная точка **Meetup RSVP** не обрабатывает сообщения, которые мы отправляем, но, просто для удовольствия, мы можем отправить текстовое сообщение:

```
webSocket.sendText("Я - поклонник Meetup RSVP", true);
```

Булев аргумент используется для обозначения конца сообщения. Если это иницирование не завершается, то сообщение передает `false`.

Для закрытия соединения нам нужно использовать метод `sendClose()`, как показано ниже:

```
webSocket.sendClose(WebSocket.NORMAL_CLOSURE, "ok");
```

Наконец, нам нужно написать слушателя `WebSocket.Listener`, который будет обрабатывать входящие сообщения. Он представляет собой интерфейс, который содержит связку методов с имплементациями по умолчанию. Следующий ниже исходный код просто переопределяет методы `onOpen()`, `onText()` и `onClose()`. Склеив `WebSocket`-слушателя и предыдущий исходный код, мы получим такое приложение:

```
public class Main {
    public static void main(String[] args) throws
        InterruptedException, ExecutionException, TimeoutException {

        Listener wsListener = new Listener() {
            @Override
            public CompletionStage<?> onText(WebSocket webSocket,
                CharSequence data, boolean last) {
                System.out.println("Получены данные: " + data);

                return Listener.super.onText(webSocket, data, last);
            }

            @Override
            public void onOpen(WebSocket webSocket) {
                System.out.println("Соединение открыто...");
                Listener.super.onOpen(webSocket);
            }

            @Override
            public CompletionStage<?> onClose(WebSocket webSocket,
                int statusCode, String reason) {
                System.out.println("Закрытие соединения: " +
                    statusCode + " " + reason);

                return Listener.super.onClose(webSocket, statusCode, reason);
            }
        };

        HttpClient client = HttpClient.newHttpClient();

        WebSocket webSocket = client.newWebSocketBuilder()
            .buildAsync(URI.create("ws://stream.meetup.com/2/rsvps"), wsListener)
            .get(10, TimeUnit.SECONDS);

        TimeUnit.SECONDS.sleep(10);

        webSocket.sendClose(WebSocket.NORMAL_CLOSURE, "ok");
    }
}
```

Это приложение будет работать в течение 10 секунд и выдаст результат, подобный показанному ниже:

Соединение открыто...

Получены данные:

```
{"visibility":"public","response":"yes","guests":0,"member":{"member_id":267133566,"photo":"https://secure.meetupstatic.com/photos/member/8/7/8/a/thumb_282154698.jpeg","member_name":"SANDRA MARTINEZ"},"rsvp_id":1781366945...
```

Получены данные:

```
{"visibility":"public","response":"yes","guests":1,"member":{"member_id":51797722,...
```

...

Через 10 секунд приложение отключается от веб-сокетной конечной точки.

Резюме

Наша работа сделана! Эта задача была последней в настоящей главе. Итак, мы подошли к концу этой книги. Похоже, что новые API HTTP-клиента и протокола WebSocket — крутые инструменты. Они обладают значительной гибкостью и универсальностью, являются довольно интуитивными, и им удастся успешно скрывать множество болезненных деталей, с которыми мы не хотим иметь дело во время разработки.

Вы можете скачать приложения из этой главы, чтобы увидеть результаты и получить дополнительные детали, по адресу <https://github.com/PacktPublishing/Java-Coding-Problems>.

Предметный указатель

A

- API:
- ◊ веб-сокеты, задачи 670
 - ◊ потоков, выстраивание в цепочку 666–667
 - ◊ рефлексии Java:
 - аннотации 389–395
 - для создания экземпляров классов:
 - из файлов JAR 373
 - посредством приватного конструктора 372
 - задачи 359
 - инициирование экземплярных методов 395–396
 - инспектирование аннотаций:
 - выбрасываемых исключений 391
 - интерфейсов 394
 - классов 390
 - методов 390
 - модулей 402–404
 - надклассов 393
 - пакетов 389
 - параметров методов 392
 - полей 393
 - типов возвращаемых значений 392
 - получение:
 - аннотации объявленной 395
 - аннотации по типу 394
 - имени класса Pair посредством экземпляра 366
 - имени некоторого типа 369
 - имплементированных интерфейсов класса Pair 366
 - класса для индексируемого типа компонента классом Pair 371
 - конструкторов 367
 - методов класса Pair 368
 - методов статических 396–397
 - модификаторов класса Pair 366
 - модуля класса Pair 369
 - надкласса класса Pair 369
 - полей класса Pair 367
 - полей приватных 400–401
 - полей публичных 400–401
 - строк, описывающих класс 370
 - строки с дескриптором типа для класса 370
 - типа компонента для массива 370
 - посредники динамические 404–407
 - работа с массивами 401–402
 - решения:
 - аннотация типа получателя 374–375
 - для геттеров 382–389
 - для сеттеров 382–389
 - для создания экземпляров классов посредством отрефлексируемого конструктора 371–374
 - инспектирование классов 365–371
 - инспектирование пакетов 361–365
 - мостовая конструкция 376–77
 - проверка дефолтных методов 378
 - проверка переменного числа аргументов 377–78
 - синтетическая конструкция 376–77
 - управление гнездовым доступом (гнездами) 378–82
 - ◊ фрагменты кода 374

B

BigInteger 73–74
boolean 668–669
BufferedReader против Scanner 357

C

Callable, интерфейс 553–558
◊ операция 553–558
Cloning, библиотека 119
CompletableFuture, класс 596–611
◊ exceptionallyCompose() в JDK 12 607
◊ завершение явным образом 610
◊ операция асинхронная 596–598
 ▫ обработка исключений посредством:
 • exceptionally() 602, 603, 605
 • handle() 608
◊ сочетание 611–16
 ▫ посредством allOf() 613
 ▫ посредством anyOf() 615
 ▫ посредством thenCombine() 612
 ▫ посредством thenCompose() 611
◊ функция обратного вызова 600, 601
CountedCompleter 590

D

Date, конвертирование в:
◊ DateLocalTime 158
◊ Instant 156
◊ LocalDate 157
◊ LocalTime 159
◊ OffsetDateTime 159
◊ OffsetTime 159
◊ Temporal 156–60
◊ ZonedDateTime 158
double 71–72
Dual-pivot quicksort 202
Duration, класс 142

F

FileVisitor, связка методов 296–304
float 71–72

Future, интерфейс 553–558
◊ отмена операции 557

G

GMT 146–48
◊ временной пояс 147
Gson, библиотека 328

H

HashMap, 251
HTTP:
◊ метод 678–679
◊ клиент, задачи 670
HTTP/2, протокол 672–673, 698–702

I

Instant, класс 137–140
◊ вычитание времени из объекта 138
◊ генерирование меток машинного времени 137–140
◊ конвертирование объектов в:
 ▫ LocalDateTime 139
 ▫ OffsetDateTime 139
 ▫ String 138
 ▫ ZonedDateTime 139
◊ прибавление времени в объект 138
◊ сравнение объектов 139

J

Jackson, библиотека 327
JAR-файл 373
JDK, встроенные решения 202
JDK 8 130
JDK 9 161
JSON:
◊ запрос нового объекта User 692
◊ клонирование объектов 121
◊ конвертирование отклика в объект User 691
◊ обновление 689–692

◇ получение 689–692
 ◇ сохранение 689–692
 JSON-B, библиотека 326

L

List:

◇ замена элементов 258
 ◇ фильтрация коллекции 256–258
 LocalDate, класс 137
 LocalDateTime, класс 137
 ◇ получение из LocalDate 137
 ◇ получение из LocalTime 137

O

Optional 648–649, 667–668
 ◇ возврат:

- значения по умолчанию:
 - несуществующего 646
 - сконструированного 645
- класса Optional 650–651
- типа boolean 668–69

 ◇ исключение NoSuchElementException 647–648
 ◇ выстраивание в цепочку 666–667
 ◇ задачи 641
 ◇ значение недостающее 644
 ◇ инициализация 643–644
 ◇ недопущение 652–660
 ◇ подтверждение эквивалентности 662–663
 ◇ потребление присутствующего класса Optional 649–650

P

Pair, класс:

- ◇ имя 366
- ◇ интерфейс имплементированный 366
- ◇ конструктор 367
- ◇ метод 368
- ◇ модификатор 366
- ◇ модуль 369

◇ надкласс 369
 ◇ поле 367
 Period, класс 140

R

RecursiveAction, вычисление чисел
 Фибоначчи 588
 RecursiveTask, вычисление суммы 587
 ReentrantLock, класс 627–630
 ReentrantReadWriteLock, класс 630–633

S

Scanner 354, 357
 Spliterator, интерфейс 504
 StampedLock, класс 633–37
 switch 123–125

T

Temporal, конвертирование в объект Date 156–160
 ThreadLocal 618–623
 ◇ контекст в расчете на нить исполнения 621
 ◇ экземпляры в расчете на нить исполнения 619

U

Union Find, алгоритм 272–277
 ◇ имплементирование операции:

- объединения 276
- поиска 275

 UTC 146–148
 ◇ временной пояс 147

V, W

var 184
 WebSocket, протокол 702–70

А

Аксессуар 382
Алгоритм сортировки 204
Аннотация:
◇ инспектирование 389
◇ объявленная 395
◇ типа получателя 374–75
Аннотирование:
◇ методов 390
◇ пакетов 389
Аутентификация соединения 681–682

Б

Банда четырех (GoF), книга 112
Барьер 564–568
Блок:
◇ инструкций 126–127
◇ статический 234
Буква:
◇ инвертирование 38–39
◇ подсчет гласных/согласных 40–41
Буфер байтовый, имплементирование 339–346

В

Ввод-вывод в среде Java, 285, 516
Возраст, вычисление 161–162
Время:
◇ машинное, генерирование временных меток 137–140
◇ проверка 498
◇ продолжительность 140–144
Вывод типа локальной переменной логический (LVTI) 172
◇ for 187–188
◇ try 194
◇ замена:
▫ ковариантов 197
▫ контравариантов 197
▫ подстановочных знаков 196
◇ избегание использования с:
▫ null-инициализаторами 194–195

▫ переменными блока catch 194–195
▫ экземплярами переменными 194–195
◇ класс анонимный 191
◇ лямбда-выражение 193–194
◇ недопущение 180–181
◇ область видимости переменной 185–186
◇ объявления составные 184–185
◇ оператор тренажный 186–187
◇ поток 188–189
◇ практически финальный 192–193
◇ примеры 174–177
◇ разбиение вложенных/крупных цепочек выражений 189–190
◇ совмещение с обобщенными типами T 195–196
◇ сочетание с:
▫ помощью программирования 182
▫ явного понижающего приведения типа 180
▫ ромбовидным оператором 183
◇ техническая сопроводимость исходного кода 178–179
◇ тип:
▫ аргумента 190–191
▫ возвращаемый из метода 190–191
▫ примитивный 177–178

Вывод типов, логический, задачи 172
Вычисление высокопроизводительное (HPC) 79

Г

Геттер 382, 385
Группирование элементов потока 478–485
◇ многоуровневое 484
◇ одноуровневое 478

Д

Данные:
◇ плохие, недопущение в немутуируемых объектах 114–116
◇ формы, обработка 693–695

Дата:

- ◇ `LocalDate` 160–61
- ◇ вычисление разницы между датами 164–166
- ◇ обход интервала в цикле 160–161
 - `Calendar` 160
 - `LocalDate.datesUntil(LocalDate.endExclusive)` 161

Дата-время:

- ◇ `Calendar` 164–166
- ◇ `Date` 145
- ◇ `java.util.Date` 164–166
- ◇ `LocalDate` 137
- ◇ `LocalDateTime` 145
- ◇ `LocalTime` 137
- ◇ `Temporal` 164–166
- ◇ без даты и времени 137
- ◇ вывод на экран полетной информации 150–151
- ◇ вычитание времени 145–146
- ◇ единица даты и времени 144–145
- ◇ конвертирование:
 - в строку 130
 - `DateFormat` 130–133
 - `DateTimeFormatter` 130
 - временной метки UNIX 151–152
- ◇ локальная, получение:
 - `Date` 149
 - `ZonedDateTime.now()` 149
 - во всех временных поясах 148–149
- ◇ прибавление времени 145–46
- ◇ форматирование 134–36

Деление:

- ◇ беззнаковых значений 70–71
- ◇ целая часть:
 - деления 75–76
 - остаток от деления 75–76

День месяца:

- ◇ первый 152–156
- ◇ последний 152–154

Дерево:

- ◇ префиксное 63, 266–271
 - вставка 268
 - отыскание 269
 - строительство 268
 - структура базовая 268

- удаление 269
- узел 267

- ◇ Фенвика (BIT) 277–281
- ◇ цифровое 266–271

3

Заголовок:

- ◇ запроса 676–678
- ◇ настройка 676–678
- ◇ получение 676–78

Замок на уровне:

- ◇ класса 524–527
- ◇ объекта 524–527

Запрос:

- ◇ GET асинхронный 673–675
- ◇ отправка:
 - асинхронно 684–686
 - синхронно 684–686

Знак подстановочный 196

Значение:

- ◇ `int`:
 - конвертирование в `long` 74–75
 - сложение 67–68
 - умножение 77–78
- ◇ `long`:
 - конвертирование в `int` 74–75
 - сложение 67–68
 - умножение 77–78
- ◇ `Optional`:
 - `flatMap()` 663–665
 - `map()` 663–665
- ◇ беззнаковое:
 - деление 70–71
 - остаток от деления 70–71
- ◇ дефолтное, возвращение из коллекции `Map` 238–239
- ◇ многострочное строковое 59–60
- ◇ на основе:
 - времени 140–144
 - даты 140–144
- ◇ псевдослучайное, создание нелимитированных потоков 453
- ◇ с плавающей точкой 76–77

И

Идентификатор ресурса

унифицированный (URI) 288

Индекс, проверка в интервале от 0 до
длины 94–96

Инспектирование аннотаций параметров
методов 392

Интервал, даты в цикле 160–161

Интернирование строк 104

Интерфейс:

- ◊ инспектирование аннотаций 394
- ◊ обобщенный 397–400
- ◊ функциональный 409–417
 - абстрагирование типа List 415
 - имплементирование фильтров 414
 - класс анонимный как лямбда-выражение 415
 - передача поведения в качестве параметра 412
 - фильтрация дынь 410, 411

Исключение:

- ◊ выбрасываемое 391
- ◊ обобщенное 397–400

К

Каркас разветвления/соединения 585–592

Класс немутуируемый:

- ◊ написание 108–109
 - посредством шаблона строителя 112–114
- ◊ объект мутируемый:
 - возврат 109–111
 - передача 109–111

Ковариант, замена с помощью
логического вывода типа локальной
переменной (LVTI) 196–197

Код исходный:

- ◊ в функциональном стиле, проверка
ссылок null 86–89
- ◊ императивный, проверка ссылок null
86–89

Коллектор:

- ◊ подытоживающий 474–77
 - максимум 477

- минимум 477
- подсчет 476
- сложение 474
- усреднение 476

- ◊ собственный 494–499
 - посредством collect() 499

Коллекция:

- ◊ задачи 199
 - ◊ конвертирование в массив 255–56
 - ◊ конкурентная встроенная в Java 259–64
 - коллекция синхронизированная 263, 264
 - множество нитебезопасное 259
 - отображение Map нитебезопасное 260
 - очередь нитебезопасная:
 - задержки 262
 - на основе связанных узлов 262
 - поддерживаемая массивом 261
 - с приоритетом 262
 - синхронная 262
 - трансферная 262
 - против встроенных в Java синхронизированных коллекций 264
 - списки нитебезопасные 259
 - стек нитебезопасный 263
 - ◊ немодифицируемая 233–238
 - Arrays.asList() 234
 - Collections.unmodifiableList() 233
 - List.of() 235
 - блок статический 234
 - ◊ немутуируемая 233–238
 - Arrays.asList() 234
 - Collections.unmodifiableList() 233
 - List.of() 235
 - создание 233
 - блок статический 234
 - ◊ нитебезопасная 259–264
 - ◊ фильтрация по списку 256–258
- Компаратор, композиция 510
- Композиция 508–513
- ◊ компараторов 510
 - ◊ предикатов 509
 - ◊ функций 512
- Конверсия беззнаковая 69
- Конец дня, поиск 162–164

Конкурентность 580

Конструктор:

- ◊ клонирование объектов 118
- ◊ отрефлексируемый 371–374
- ◊ приватный 372

Конструкция:

- ◊ мостовая 376–377
- ◊ синтетическая 376–377

Контравариант, замена с помощью логического вывода типа локальной переменной (LVTI) 196–197

Кортеж 271–272

Л

Лямбда-выражение 193–194, 417–418

- ◊ orElseFoo() 651–652
- ◊ отладка 445–448
- ◊ тестирование методов 443–445

М

Массив 401–402

- ◊ алгоритмы сортировки 204
- ◊ задачи 199
- ◊ значение:
 - максимальное 224–227
 - минимальное 224–227
 - среднее 224–227
- ◊ заполнение 229–231
- ◊ инвертирование 227–229
- ◊ конвертирование коллекции 255–256
- ◊ сравнение лексикографическое 221–223
- ◊ настройка 229–231
- ◊ тип компонента 370
- ◊ поиск элемента 218–221
- ◊ поток 223–224
- ◊ примитивных типов 203
- ◊ присвоение переменной var 184
- ◊ проверка:
 - несовпадения 218–221
 - эквивалентности 218–221
- ◊ размер, изменение 232

- ◊ решения, встроенные в JDK 201–213
- ◊ сортировка 201
 - массива строк по длине 54–56
 - параллельная 202
 - по убыванию 203

◊ целых чисел 202

Метка:

- ◊ варианта 126
- ◊ временная UNIX, конвертирование в дату-время 151–152

Метод:

- ◊ accumulator() 497
- ◊ Arrays.asList() 218
- ◊ characteristics() 498
- ◊ clone() 117
- ◊ Collections.unmodifiableList() 233, 236
- ◊ combiner() 497
- ◊ compareAndSetForkJoinTaskTag() 593–596
- ◊ dropWhile() 450–458
- ◊ equals() 97–103
- ◊ filtering() 488–491
- ◊ finisher() 497
- ◊ flatMapping() 490
- ◊ hashCode() метод 97–103
- ◊ List.of() 235
- ◊ mapping() 490
- ◊ max() 66–67
- ◊ min() 466–470
- ◊ ofNullable() 660–661
- ◊ Optional.filter() 665
- ◊ Optional.get() 646
- ◊ Optional.of() 660–661
- ◊ orElseFoo() 651–52
- ◊ Path.equals() 294
- ◊ sum()
 - в потоке 466–470
 - вычисление посредством RecursiveTask 587
 - терминальные операции 467
- ◊ supplier() 496
- ◊ takeWhile() 450–458
 - связка методов 454
- ◊ toString() 121–123
- ◊ обобщенный 397–400
- ◊ по умолчанию 513–514

- ◇ прерываемый 581–585
- ◇ статический 396–397
- ◇ тестирование с лямбда-выражениями 443–445
- ◇ трафаретный 423
- ◇ экземплярный 395–396
- Модуль, инспектирование 402–404

Н

- Наблюдатель 425–428
- Наблюдение за путями 304–307
- ◇ модифицирование папки 305
- Надкласс 393, 399
- Начало дня, поиск 162–164
- Немутируемость строковых значений 104–108
- ◇ недостатки 106
- ◇ преимущества 104

О

- Обменник 568–571
- Объект:
 - ◇ User 691, 692
 - ◇ клонирование 116–121
 - clone() 117
 - Cloning 119
 - JSON 121
 - конструктор 118
 - ручное 117
 - сериализация 120
 - ◇ мутируемый, класс немутуруемый:
 - возврат 109–111
 - передача 109–111
 - ◇ немутуруемый 103
 - недопущение плохих данных 114–116
 - ◇ обыкновенный Java (POJO) 121
- Обязанность межобъектная (сквозная) 404
- Ожидание:
 - ◇ занятое оптимизирование 616
 - ◇ циклическое 637

- Операция:
 - ◇ Callable 558–560
 - ◇ И 72–73
 - ◇ ИЛИ 72–73
 - ◇ ИЛИ исключающее 72–73
 - ◇ объединения 276
 - ◇ отмена 617–618
 - ◇ поиска 275
 - ◇ чувствительная к идентичности 667–668
- Остаток от деления 75–76
 - ◇ беззнаковые значения 70–71
 - ◇ целая часть 75–76
- Отклик:
 - ◇ заголовок 677
 - ◇ информация 687
- Отображение:
 - ◇ compute() 241
 - ◇ computeIfAbsent() 240
 - ◇ computeIfPresent() 239
 - ◇ merge() 243
 - ◇ putIfAbsent() 244
 - ◇ вычисление отсутствия/присутствия элемента 239–244
 - ◇ замена элементов 245–246
 - ◇ слияние двух отображений 252–253
 - ◇ сортировка 248–251
 - по значению посредством:
 - компаратора 249
 - потока 249
 - списка 250
 - по ключу посредством:
 - TreeMap 249
 - компаратора 249
 - потока 249
 - списка 250
 - ◇ сравнение отображений 246–248
 - ◇ удаление элемента 244–245
 - ◇ упорядочивание в естественном порядке 249
 - ◇ элементов потока 458–463
 - Stream.flatMap() 460
 - Stream.map() 458
- Отправка многочисленных запросов, конкурентная 685
- Отступ 63–65
- Очередь нитебезопасная 259–264

П

Пакет:

- ◊ инспектирование 361
 - внутри модулей 365

- ◊ получение классов 362

Папка, поиск в файловом дереве 308–310

Пара суррогатная 34

Параметризация поведения 412

Первым вошел первым вышел (FIFO) 261

Переменная атомарная 623–627

- ◊ накопители 626

- ◊ сумматоры 626

Переполнение операции:

- ◊ сложения 67–68

- ◊ умножения 77–78

Перестановка, генерирование 46–48

Перетасовка:

- ◊ Кнута 213

- ◊ массива объектов 213

- ◊ Фишера — Йейтса 213

Период времени 140–144

Подинтервал 96–97

Подстрока, подсчет появлений в строке 56–57

Поиск:

- ◊ двоичный 63

- ◊ сперва в ширину (BFS) 264–266
 - алгоритм 265
 - псевдокод 264

- псевдокод 264

Поле:

- ◊ инспектирование аннотаций 393

- ◊ обобщенное 397–400

Политика перенаправления 683

Последним вошел первым вышел (LIFO) 185

Посредник динамический 404–407

- ◊ имплементирование 405

Построитель параметров запроса 675

Поток:

- ◊ max() 466–470

- ◊ min() 466–470

- ◊ null-безопасный 506–508

- ◊ sum() 466–470

- ◊ бесконечный 450–458
 - последовательный неупорядоченный 454

- последовательный упорядоченный 451

- ◊ нелимитированный 453

- ◊ обработка параллельная 501–506

- ◊ операция терминальная 466–470

- ◊ поиск элемента 463–465

- ◊ сбор результатов 470–473

- ◊ соединение результатов 473–474

- ◊ создание из массива 223–224

- ◊ сопоставление элементов 465–466

- ◊ фильтрация ненулевых элементов 448–450

Пояс часовой:

- ◊ API даты-времени Java 146–148

- ◊ GMT 146–148

- ◊ UTC 146–148

- ◊ дата-время локальное 148–149

- ◊ извлечение смещения 147

Предикат 412

- ◊ композиция 509

Предмет в шаблоне наблюдателя 425

Префикс наибольший общий 62–63

Программирование в функциональном стиле 408, 439

Прокси-сервер 675–676

Прохождение путей 296–304

- ◊ File.walk() 303

- ◊ JDK 8 303

- ◊ копирование папки 301

- ◊ поиск файла по имени 297

- ◊ тривиальный обход папки 297

- ◊ удаление папки 299

Пул:

- ◊ нитей исполнения 527–531
 - Executor 527
 - Executors 530
 - ExecutorService 527
 - ScheduledExecutorService 530
 - одна нить 531–539
 - производитель ждет наличия потребителя 532
 - производитель не проверяет наличия потребителя 537
 - с запланированными нитями 540–547
 - с кэшированными нитями 540–547

- с фиксированным числом нитей 539–340
- обкрадывающих 547–553
 - крупное число малых операций 549
 - малое число времязатратных операций 551

◇ строковых констант (SCP) 104

Путь:

- ◇ абсолютный 290
- ◇ к файлу:
 - абсолютный 289
 - конвертирование 291–292
 - конструирование 293–294
 - сравнение лексикографическое 295
 - представление одинакового файла/папки 295
 - соединение 292–293
 - создание 287–291
 - в корне файлового хранилища 288
 - в текущей папке 288
 - с помощью сокращений 289
 - сравнение 294–296
 - Path.equals() 294
 - частичное 296

Р

Разбиение 486–488

Разграничитель 44–45

Редукция 468

Ресурс:

- ◇ закачивание на сервер 696–698
- ◇ скачивание с сервера 695

С

Семафор 571–574

Сериализация, клонирование объектов 120

Сеттер:

- ◇ генерирование 385
 - ◇ получение 382
- Сжатие 692–93

Символ:

- ◇ идеографический (CJKV) 506
 - ◇ отыскание первого неповторяющегося символа 36–38
 - ◇ подсчет дубликатов 32–35
 - ◇ удаление 51–52
 - дубликатов символов 49–51
 - ◇ юникодный 33
- Синхронизация 526
- Слово, инвертирование 38–39
- Смещение поясное, извлечение 154–156
- ◇ java.time.ZoneId() 155
 - ◇ java.time.ZoneOffset() 155
 - ◇ TimeZone.getRawOffset() 155
- Сокращение 289
- Сортировка:
- ◇ вставками 206
 - ◇ кучи 209
 - ◇ массива:
 - параллельная 202
 - по убыванию 203
 - ◇ подсчетом 208
 - ◇ пузырьком 204
 - ◇ с двойным опорным элементом 202
- Состояние жизненного цикла нити исполнения Java 518–524
- ◇ BLOCKED 520
 - ◇ NEW 519
 - ◇ RUNNABLE 519
 - ◇ TERMINATED 523
 - ◇ TIMED_WAITING 522
 - ◇ WAITING 521
- Сочленение 492–494
- Сплитератор собственный 506
- Сравнение, подсчет 40–41
- Сравнить и поменять местами (CAS Compare и Swap) 624
- Ссылка null 648–49
- ◇ Optional 648–49
 - ◇ на метод 499–501
 - статический 500
 - экземплярный 500
 - ◇ на конструктор 501
 - ◇ проверка:
 - в императивном стиле 86–89
 - в функциональном стиле 86–89
 - и возврат дефолтных ссылок не-null 93

- и выбрасывание заданного исключения 91–92
- и выбрасывание специализированного исключения
NullPointerException 89–91
- Стек нитебезопасный 259–264
- Стопор 560–564
- Строка:
 - ◇ генерирование перестановок 46–48
 - ◇ задачи 29
 - ◇ инвертирование:
 - символов 38–39
 - слов 38–39
 - ◇ конвертирование в:
 - double 43–44
 - float 43–44
 - int 43–44
 - long 43–44
 - дату-время 130–133
 - ◇ конкатенация 60–62
 - ◇ отступ 63–65
 - ◇ подсчет:
 - гласных 40–41
 - повторяющихся символов 32–35
 - появлений некоторого символа 42–43
 - появлений подстроки 56–57
 - согласных 40–41
 - ◇ поиск:
 - наибольшего общего префикса 62–63
 - первого неповторяющегося символа 36–38
 - символа с наибольшим числом появлений 52–54
 - ◇ проверка на наличие:
 - анаграмм 58–59
 - палиндрома 48–49
 - подстроки 56
 - цифр 39–40
 - ◇ решения 32, 86, 130, 174, 201, 287, 361, 409, 441, 518, 581, 643, 672
 - ◇ соединение с помощью разделителя 44–45
 - ◇ сортировка массива по длине строк 54–56
 - ◇ трансформирование 65–66

- ◇ удаление:
 - пробелов 44
 - замыкающих 62
 - начальных 62
 - символов 51–52
 - повторяющихся 49–51
- ◇ число беззнаковое с основанием системы счисления 68–69
- Структуры данных, задачи 199

Т

- Таймаут 682–683
- Тело запроса:
 - ◇ настройка 679–681
 - ◇ создание из:
 - InputStream 680
 - значения типа String 679
 - массива байт 680
 - файла 681
- Тело отклика, обработка 687–689
- ◇ как InputStream 688
- ◇ как значения типа String 688
- ◇ как массива байтов 688
- ◇ как потока значений типа String 689
- ◇ как файла 688
- Тип возвращаемого значения, инспектирование аннотаций 392
- Тупик 637–40

У

- Умножение и сложение совмещенное (FMA) 79
- Управление гнездовым доступом (гнездами) посредством API рефлексии Java 378–382

Ф

- Фазировщик 574–579
- Файл:
 - ◇ cookie 686–687
 - ◇ CSV 325–329
 - ◇ JSON 325–329

- ◇ большой
 - `BufferedReader` 322
 - `MappedByteBuffer` 324
 - `Scanner` 323
 - поиск строки в файле 321–325
 - `Файл.lines()` 323
 - `Файл.readAllLines()` 322
- ◇ временный 330–334
 - создание 330
 - удаление посредством:
 - `DELETE_ON_CLOSE` 333
 - `deleteOnExit()` 332
 - перехватчика отключения 331
- ◇ двоичный:
 - запись 320
 - эффективная 317
 - чтение:
 - в память 319
 - эффективное 317–321
- ◇ запись форматированного результата 351–54
- ◇ лексемизация (токенизация) 347–51
- ◇ многочастный, закичивание на сервер 696–698
- ◇ обнаружение несовпадений 337–339
- ◇ передача потоковая 307–308
- ◇ поиск в деревьях файлов 308–310
- ◇ текстовый:
 - запись 316
 - эффективная 310–317
 - чтение:
 - в память 314
 - эффективное 310–317
- ◇ фильтрация 334–337
 - `File.newDirectoryStream()` 334
 - `FileFilter` 337
 - `FilenameFilter` 336
- Фибоначчи последовательность 588
- Фильтр Блума 281–284
- Функция:
 - ◇ высшего порядка 441–43
 - ◇ композиция 512

Ч

Часы шахматные 166–171

Число:

- ◇ беззнаковое, сравнение 70

- ◇ компактное:
 - разбор 82
 - форматирование 79–83
- ◇ максимум 66–67
- ◇ минимум 66–67

Ш

Шаблон:

- ◇ декоратора 430–434
- ◇ исполнения вокруг опорной операции 418–420
- ◇ каскадного строителя 434–435
- ◇ команд 435–438
- ◇ наблюдателя 425–428
- ◇ одалживания 428–430
- ◇ стратегий 422–423
- ◇ строителя, класс немутуруемый 112–114
- ◇ трафаретного метода 423–425
- ◇ фабричный 420–422
- ◇ форматирования даты-времени 134–136

Э

Элемент:

- ◇ замена в списке 258
- ◇ не-null, фильтрация из потока 448–450
- ◇ поиск:
 - в массиве 218–221
 - в потоке 463–465
 - `findAny()` 464
 - `findFirst()` 464
- ◇ проверка:
 - присутствия 214
 - только первого индекса 217
- ◇ разделение посредством `Collectors.partitioningBy()` 255
- ◇ следующий больший (NGE) 231
- ◇ сопоставление в потоке 465–466
- ◇ удаление:
 - из коллекции 253–255
 - посредством `Collection.removeIf()` 254
 - посредством `Stream` 254
 - посредством итератора 2

Java РЕШЕНИЕ ПРАКТИЧЕСКИХ ЗАДАЧ

В книге приведено более 300 приложений, содержащих свыше 1000 примеров решения типовых задач, с которыми приходится иметь дело каждому разработчику в среде Java.

Продемонстрированы эффективные практические приемы и технические решения с учетом сложности кода, производительности, удобочитаемости и многого другого.

Рассмотрены строки, числа, объекты, массивы, коллекции и структуры данных, работа с датой и временем.

Приведены задачи на логический вывод типов, а также файловый ввод/вывод.

Представлены задачи, связанные с API рефлексии Java.

Особое внимание уделено программированию в функциональном стиле:

рассмотрены как основы и шаблоны архитектурного дизайна, так и вопросы, требующие глубокого погружения в тему, — например, отладка лямбда-выражений.

Рассмотрены основополагающие задачи на тему конкурентности, а также углубленные задачи на темы разветвления/соединения, атомарных переменных, прерываемых методов и др.

Несомненный интерес представляют задачи на правила работы с классом Optional, а также API HTTP-клиента и API протокола WebSocket.

Прочитав эту книгу, вы получите глубокое понимание концепций среды Java и обретете уверенность при разработке приложений и выборе правильных решений своих задач.

По ходу чтения книги вы:

- Научитесь применять новейшие средства JDK 11 и JDK 12 для разработки своих приложений
- Решите актуальные задачи, связанные с коллекциями и структурами данных
- Освоите программирование в функциональном стиле с использованием лямбда-выражений
- Выполните асинхронную передачу и параллельную обработку данных
- Решите задачи со строками и числами с помощью новейших API Java
- Познакомитесь с разными аспектами немутуируемости объектов в среде Java
- Научитесь использовать правильные практические приемы и эффективные методы программирования

Packt>

www.packt.com



191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

ISBN 978-5-9775-6719-0



9 785977 567190